NEKO series

猫でもわかる C言語プログラミング

第2版



粂井康孝 著
Yasutaka Kumei

SoftBank Creative

NEKO series

猫でもわかる

第2版

C言語プログラミング

粂井康孝 著
YasutaKa Kumei

SoftBank Creative

はじめに

本書は、C言語やプログラミングそのものがまったく初めてという方を対象にした入門書です。コンピュータの動作の仕組みから始め、変数、関数、ポインタその他、C言語のプログラミングをひととおりマスターできるまで、やさしくわかりやすくレクチャーします。

C言語を修得するとどのようなプログラムが作れるようになるのでしょうか。答えは「あらゆるプログラム」です！　市販されているほとんどのソフトウェアはC言語で書かれていると言っても、言いすぎではないでしょう。そればかりではありません。WindowsやLinuxなどのOS自体も、実はC言語で書かれているのです。

また、C言語を知っていると、他のプログラミング言語の修得が非常に容易になります。C++の基本的文法はC言語のそれとまったく同じです。JavaやC#などの言語もかなり似通っています。またPHPやPython、Luaなどの新しいスクリプト言語も、書き方自体はC言語とほとんど同じです。C言語を知らずにこれらの言語を学習しようとすると、たいへんな困難を伴うでしょう。C言語は、あらゆるプログラミング言語の基本と言っても過言ではありません。そのC言語の基礎の基礎を、本書ではプログラミング経験のない方にもわかりやすいよう、懇切丁寧に解説してあります。

本書を読むにあたって、いくつかのお願いがあります。それは、単にざっと読んだだけで理解したつもりにならないでください。必ず自分でソースコードを書いて、コンパイルして実行してください。最初は、本書のプログラムを丸写ししても結構です。次に、自分であれこれプログラムを改変してみてください。そしてどうなるか実験してみてください。実践することで、理解がさらに深まります。

また、各章の練習問題にもチャレンジしてください。本文のサンプルコードよりさらに進んだ内容にも触れていますので、理解を深める手助けになるはずです。練習問題の解答がついていない、と言わないでください！　ちゃんと付録CDに収められています。今回の改訂にあたり、練習問題はすべて新しく作り直し、難易度や面白さをさらに改善しました。

「ここのところをこうしたらどうなるんだろう？」とか「ここのところをこういうふうに書くことができるのかな？」という疑問が湧くことがあると思います。でも

これは、自分で実際に試してみればすぐにわかることです。たとえプログラムが暴走しても、パソコンが壊れることはありません。ともかく、疑問に思ったことは実験してみることが大切です。それがプログラミング上達の近道です。

C言語そのものの文法事項はそれほど多くありません。本書程度の入門書は1週間くらいで読破することを目標にがんばってください。あまりゆっくり読んでいると、読み終えたころには最初のほうのことを忘れてしまいますよ！

粂井 康孝

付録CD-ROM

付録CD-ROMには、本書で使用したサンプルプログラムと開発環境などのソフトウェアを収録しています。ソフトウェアをご利用になる前には、必ずソフトウェア添付のドキュメントをご参照ください。

●CD-ROMのフォルダ構成

Sample	サンプルコード
Answer	練習問題の解答
WWW	関連Webサイト

●使用に関する注意事項

・本CD-ROMの収録データに関する著作権、およびその他のすべての権利は、著作権者に帰属します。

・本CD-ROMまたは収録データに関して、ソフトバンク クリエイティブ株式会社（以下、「当社」という）は、いかなる保証も行いません。

・本CD-ROMの使用に起因するユーザーのあらゆる損害に関し、当社は一切の責任を負いません。

・本CD-ROMおよびその収録データに関し、その使用方法もしくは非動作時の質問、サポートなどの問い合わせについて、当社は回答の義務を負わないこととします。

目次

第1章 プログラムのしくみ

本書を手にされているのは、これからC言語を始めようとしている方、以前C言語に挑戦したが途中で挫折してしまった方、Visual BasicやJavaなどの言語は知っているがC言語は初めての方などさまざまであると思います。

いずれにせよ、プログラミングを始めるにあたり、プログラムがどのようにコンピュータ上で動作するのか、その大まかな原理を知っておく必要があります。

また、コンピュータの内部では2進数が使われています。2進法についても知っていると、のちのちプログラミングの役に立ちます。「そんなこと、もう知っているよ」という方はこの章をとばして読み進めてもかまいません。

「何か、いきなり難しそうだなーー」と思われた方、心配には及びません。この本のタイトルを思い出してください。そうです。「猫でもわかる」んです。

1.1 コンピュータのしくみ

皆さんはコンピュータの機能についてご存知でしょうか。「いまさら、何をいっているのだ！」とおしかりを受けそうですが、コンピュータの機能についてこれをまとめて表現するのは意外に難しいものです。

昔から、コンピュータには次の機能があるといわれてきました。

1. 制御機能
2. 演算機能
3. 記憶機能
4. 入力機能
5. 出力機能

これを**コンピュータの５大機能**といいます。これらの機能を実現する装置をそれぞれ、制御装置、演算装置、記憶装置、入力装置、出力装置と呼びます（「機能」を「装置」に置き換えただけ）。

制御機能とはコンピュータの各装置を制御する働きです。これにより、各装置は正常に動作することができます。演算機能とはその名の通り演算（計算）をする機能です。この２つの機能が特に重要で、これは主として**CPU**（Central Processing Unit）により実現されています。

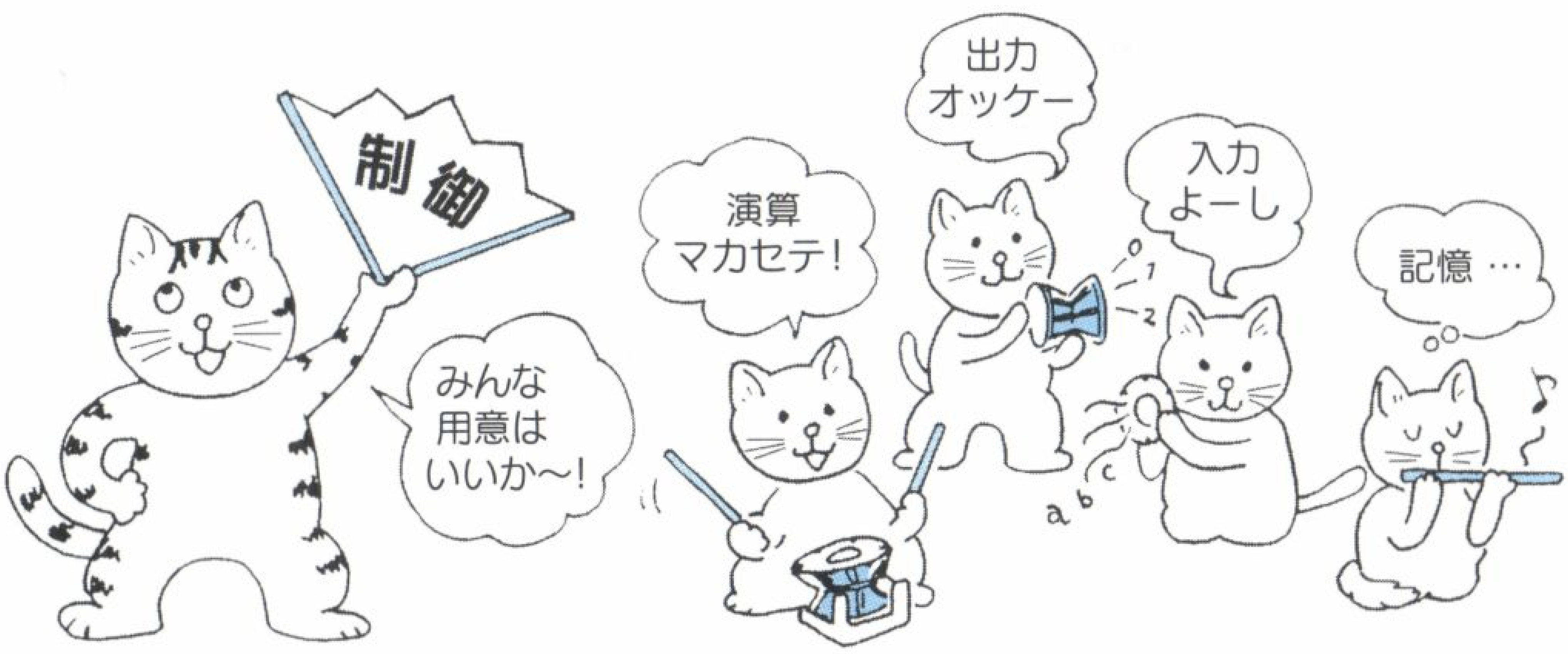
記憶機能とは、演算の結果やデータを記憶する機能です。プログラムも、実行するときには必ず記憶装置に記憶させてから実行が行われます。

入力機能とは、コンピュータに情報を伝える機能で、出力機能とは、演算の結果などを利用者に伝える機能です。

さて、CPUはどのような仕組みになっているのでしょうか。CPUは制御装置（control unit）、演算装置（ALU, Arithmetic Logic Unit）のほかにレジスタ（register）、クロック（clock）などからなります。

ほとんどすべてのコンピュータでは、プログラムは、必ずメモリに読み込まれてから実行されます。これを、**フォン・ノイマン型コンピュータ**といいます。

さて、プログラムがメモリのX番地以降に置かれたとします。すると、OS



コンピュータの5大機能

はレジスタの一種であるプログラムカウンタをXに設定します。そしてCPUがX番地の命令を実行すると、プログラムカウンタはX＋1に設定されます。すると、今度はCPUが(X＋1)番地の命令を実行します。この繰り返しでプログラムが実行されていきます。

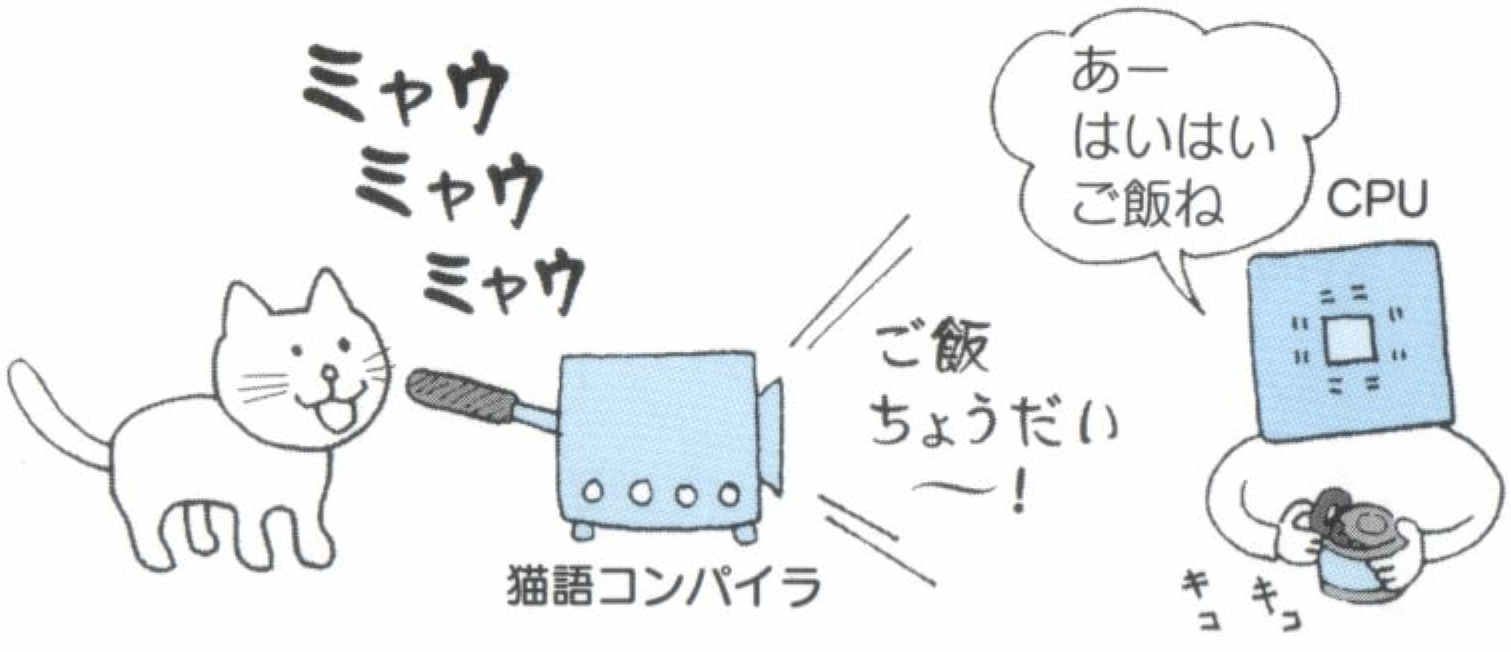
1.2 プログラムができあがるまで

プログラムが実際に実行されるときには、メモリ中にプログラムがコピーされると書きました。プログラムとは、仕事の手順をCPUに指示するものです。その手順がメモリにコピーされるのです。これから皆さんが始めるプログラミングとは、このメモリ中にコピーされるプログラムを作ることなのです。

しかし、CPUが理解できる命令は機械語命令です。この機械語命令を人間が手作業で組み合わせてプログラムを作っていたのでは、時間がいくらあっても足りません。また、間違いも頻発するでしょう。

そこで、機械語命令を人間の言葉に近い単語で表す方法が考えられました。機械語命令と1対1で対応しているのがアセンブリ言語と呼ばれているものです。

たとえば、「メモリの何番地のデータを何々のレジスタに置きなさい」といった命令を単語で置き換えるものです。これは、単に機械語を置き換えただけな



CPUを働かせるには命令をCPUが理解できる
言葉に翻訳する必要がある

ので、CPUやメモリ、そのコンピュータ独自の仕様について熟知している必要があります。

C言語のような高級言語では、たとえば画面に表示する命令をメモ帳などのテキストエディタでprintf("abc"); などと書きます。これで画面に「abc」と表示されるのです。これは、かなりわかりやすいですね。printは印字するという英語なので、printfも何かそれに近いことをするんだなとわかります。

もちろんこれだけでは、単にメモ帳で文章を作ったのと同じで、CPUに命令を出すことはできません。これを機械語に翻訳する必要があります。

メモ帳のようなテキストエディタで書いたプログラムを機械語に翻訳するソフトウェアを**コンパイラ**(compiler)といいます。また、printf("abc"); と書いてあるテキストファイルを**ソースファイル**(source file)とか単に**ソース**と呼びます。また、テキストファイルにprintf("abc"); などと書く作業を「プログラムを書く」といいます。また、「printf("abc");」など、ソースファイルに書かれている命令等を**コード**(code)と呼びます。ソースファイルを書くことを**コーディング**(coding)ともいいます。

さて、printf("abc"); と書かれたテキストファイルをコンパイラで機械語に翻訳すると、CPUに命令を出すことができるようになるのでしょうか。

実はできません。ソースファイルをコンパイルすると**オブジェクトファイル**(object file)というものが生成されます。これは、単に「printf関数を呼び出しなさい」という命令を機械語に翻訳しただけで、printf関数がどんなことをすればよいのかは、ソースファイルには書かれていません。関数という言葉が

出てきましたが、これを「命令」と置き換えて考えてください(C言語では命令を関数の形で表します)。

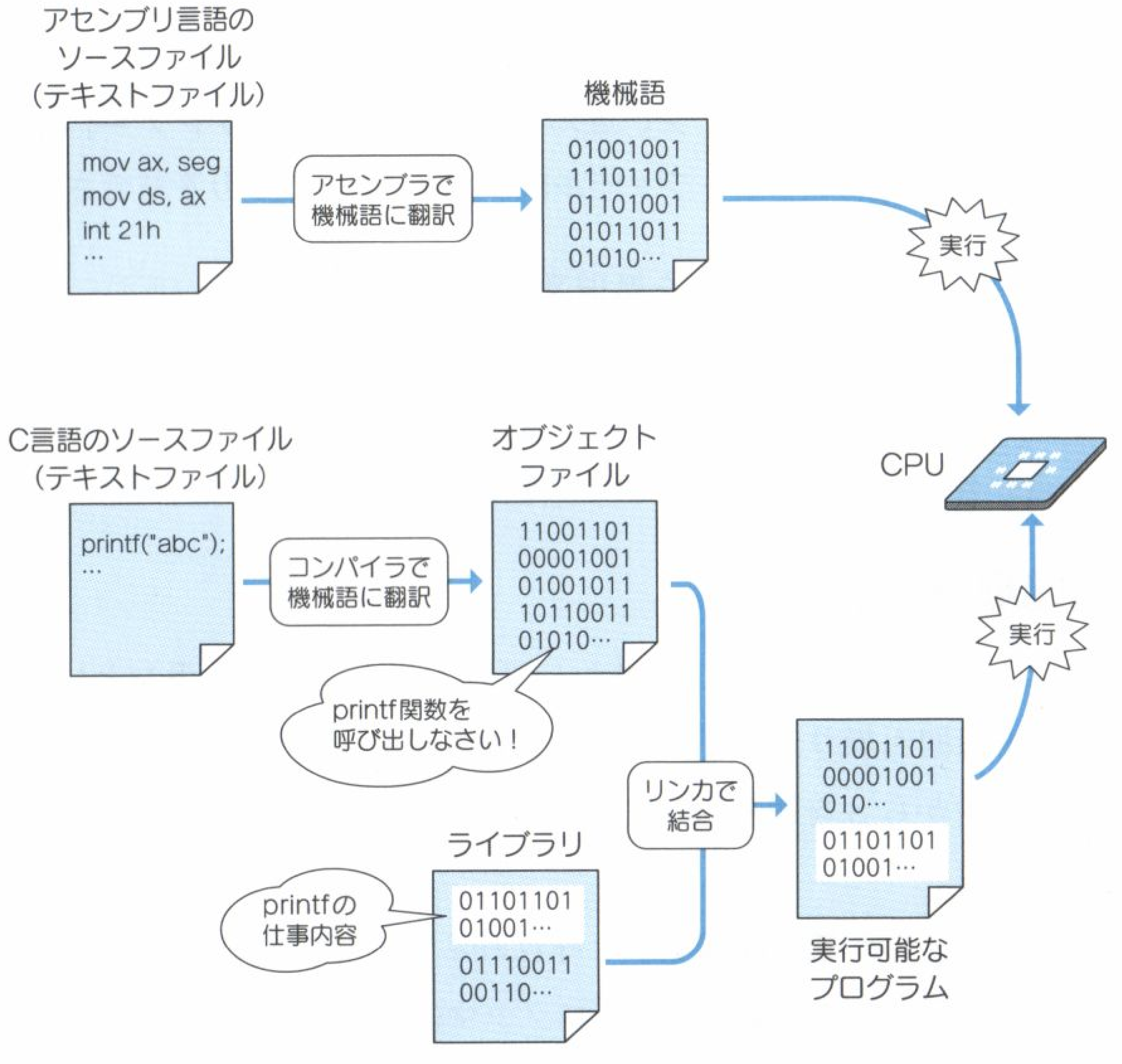


図 1-1　**プログラムができあがるまで**

printf関数がどんなことをするのかをプログラマがいちいちプログラムに書いていたのでは、これまた時間がかかってしようがありません。アセンブリ言語でプログラムを書くのと大差がなくなります。そこで、printf関数が実際に行う仕事の内容をあらかじめ機械語に翻訳して保管しているファイルがあるのです。これを**ライブラリ**(library)といいます。オブジェクトファイルには「printf関数を呼び出しなさい(printf命令を実行しなさい)」と機械語で書いてあります。ライブラリには、printf関数の実際の仕事の手順が機械語で書かれています。

ライブラリから、printfの実際の手順の機械語をオブジェクトファイルにくっつけてやる必要があります。これで、初めてCPUに命令を出せるプログラ

ムとなります。オブジェクトファイルに、ライブラリ中の機械語を埋め込むプログラムを**リンカ**(linker)といいます。

さて、C言語でプログラムを作る手順をまとめると次のようになります。

1. ソースファイルにプログラムを書く(printf("..."); など)。
2. ソースファイルをコンパイラでコンパイルし、オブジェクトファイルを作る。
3. できあがったオブジェクトファイルにリンカでライブラリを結合させる。

ところで、コンパイラのことを**処理系**(implementation)と呼ぶことがあります。**処理系依存**という言葉もよく使われます。これは、「そのコンパイラに処理が依存している」という意味です。裏を返すと、他のコンパイラでは別な処理が行われるかもしれない、という意味です。

統合開発環境とデバッグ

プログラムを少し作って試しに動かしてみることを**テストラン**といいます。テストランを何回も繰り返してプログラムが完成しますが、テストランのたびに上記の手順1から3を繰り返していたのでは非常に面倒です。プログラムに集中できません。

そこで登場したのが**統合開発環境**と呼ばれるものです。これは、コンパイラだのリンカだのを意識させません。プログラマはコーディングに集中できます。また、関数の仕様がわからなければ、ワンタッチでその関数のヘルプも見られるような工夫がなされています。

さて、作ったプログラムが思い通り動かないことは日常茶飯事です(筆者の場合はね。他の人は知らない)。これは、自分の書いたプログラムに問題があるからです。プログラムの誤りを**バグ**(bug)といいます。これには2通りあります。

1つは**文法の間違い**(syntax error)です。これは、コンパイル段階でコンパイラが指摘してくれるので、比較的簡単に発見することができます。

もう1つは、**論理的な誤り**(logical error)です。プロクラムに文法的な誤りがないため、コンパイラは誤りを指摘できません。たとえば、長方形の面積を求めるプログラムを作ったとします。「面積は縦かける横で求める」とすべきところ、誤って「縦たす横」とプログラムに書いてしまった場合などがこれに相当

します。

バグを発見する作業を**デバッグ**(debug)といい、**デバッガ**(debugger)と呼ばれるソフトウェアを使います。たとえば、計算の途中の値がわかれば、どこに間違いがあるかを発見しやすくなりますね。

さて、できあがったプログラムはすべて、機械語になっています。機械語とC言語のようなプログラミング言語には1対1の対応がありません。したがって、C言語で書いてあるソースプログラムの「この部分のこの変数の値はどうなっているのか」ということを簡単に知ることは不可能に近いのです。

そこで、ソースプログラムをコンパイルする段階で、**デバッギングコード**というものを埋め込みます。そうすると、ソースプログラム段階での、「この部分のこの変数の値」というものがわかるようになります。これを、**ソースレベルのデバッグ**といいます。

デバッギングコードを含んだプログラムは、ファイルサイズがかなり大きくなります。また、実行速度も遅いので、デバッグが終わったらデバッギングコードを含まないプログラムを作ってやる必要があります。

1.3 ビットとバイト

ビットとは

「ビット(bit)」とか「バイト(byte)」という言葉を聞いたことがない人はいないでしょう。「ビット」とは何でしょうか。**ビット**とは情報量の最小の単位です。「あーー！やっぱり難しい！！」いいえ、簡単です。最小なので「ちビット」と覚えてください。

筆者の住んでいるところには「L特急」というJRの電車が走っています。この「L特急」には、偶数車両にトイレがついています。そして、トイレが使用中の場合は「使用中」のランプが点灯して、誰かがトイレを利用していることがわかります。

「使用中」のランプの状態は「点灯」と「消灯」の２つしかありません。

このランプを見た人は、この２つの状態のいずれであるかを知ることができます。これが１ビットの情報量なのです。

「使用中」のランプが点灯していても、使用しているのが女性なのか男性なのかはわかりません。大人なのか、子供なのかもわかりません。ましてや、山田さんが利用しているのか田中さんが利用しているのか、はたまた鈴木さんが利用しているのかなどは一切わかりません。

さて、もしランプが２つあったらどうでしょうか。仮にこのランプにA，Bという名前をつけます。ランプが点灯している状態を1、消灯している状態を0で表すとすれば表1-1のような４種類の組み合わせがあることがわかります。

表 1-1　２つのランプの状態の組み合わせ

ランプA	ランプB
0	0
0	1
1	0
1	1

もし、「使用中」のランプが２つついている車両が存在するとしたら、これを見た人は４つの状態のいずれであるかを知ることができます。これが２ビットの情報量です。

ランプAとBの状態を(A，B)と書くとします。

- (0，0)がトイレ未使用
- (0，1)が男性使用中
- (1，0)が女性使用中
- (1，1)は性別不明の人が使用中

というような約束を決めることもできます。

さらに(0，0)は0、(0，1)は1、(1，0)は2、(1，1)は3というように番号をつけると、トイレの使用状況は番号で表すことも可能ですね。

では、ランプが3つあったらどうでしょうか。3つあれば8種類の状態を表すことができますね。4つでは16種類、5つでは32種類の状態を表すことができます。一般にランプがn(nは1, 2, 3, …)個あれば、2^n(2のn乗)種類の状態を表すことができます(図1-2参照)。



ビット(bit)とは、コンピュータが扱う情報の最小単位で、1ビットで2種類の状態を表現できる(nビットで2^n種類)。

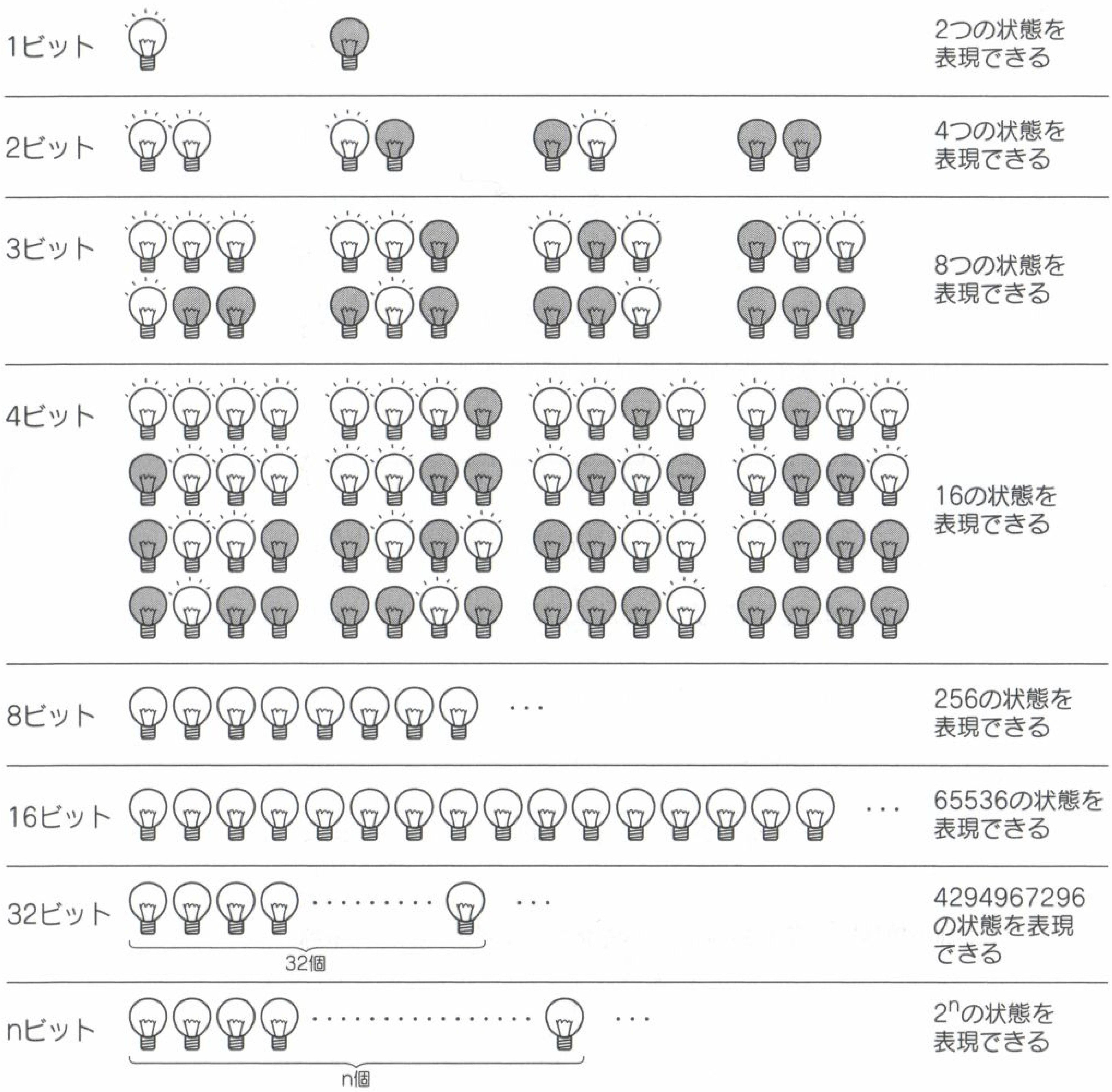


図1-2　nビットの情報量

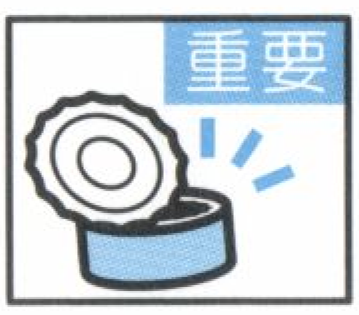
バイトとは

さて、8ビットの情報量を1**バイト**と呼びます。1バイトでは何種類の状態を表すことができるでしょうか。2の8乗で256種類ですね。ワープロでは「1バイト文字」とか「2バイト文字」という用語をよく使いますね。俗にいう半角文字が1バイト文字です。1バイトあれば、数字の0～9, アルファベットのa～zとA～Z、カタカナのア～ン、その他ちょっとした記号類にダブらないように適当な番号をつけることができます。番号は256種類もあれば十分ですね。

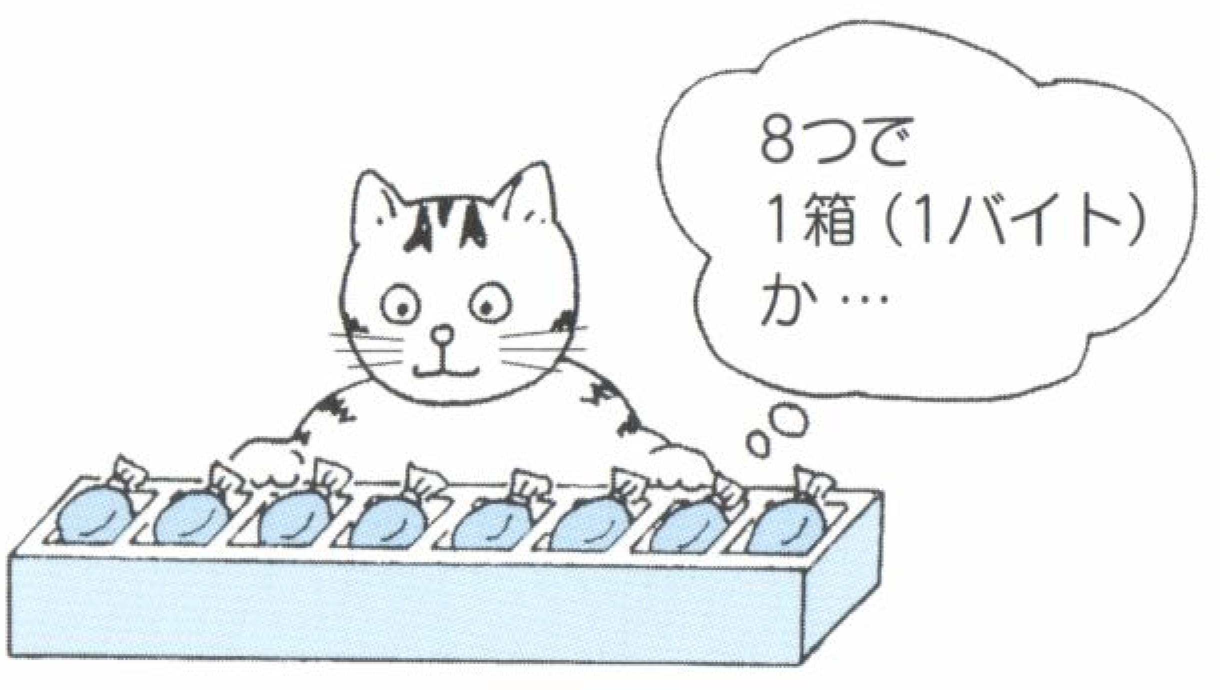
しかし、漢字はどうでしょうか。漢字に番号をつけて区別するには1バイトではまったく足りません。では、2バイトでは何種類の区別がつくのでしょうか。

2バイトは16ビットです。2^{16}は65536です。漢字に番号をつけて識別するには2バイトあればなんとかなりそうですね。それで、漢字のことを2バイト文字などと呼びます。

いずれにせよ、文字も番号をつければ数字で表すことができます。そして、その番号は0と1のみで表すことも可能ですね。



1バイトとは8ビットの情報量を表す。



1.4 2進数の基礎

私たちは日常生活において10進数の数字を使っていますね。ところで10進数って何でしょうか。「そんなの小学生でも知っているよ！」はい、すいません。10進数では数字は0から9までの10種類が存在し、9の次は桁が繰り上がって10となります。たとえば、10進数で321とはどういうことかというと

100の位は3‥‥3×100	（10の2乗）＝	300
10の位は2‥‥‥2× 10	（10の1乗）＝	20
1の位は1‥‥‥‥1× 1	（10の0乗）＝	1
合計		321

ですね。これは小学校の低学年で習いました。

では、**2進数**(binary digit)ではどうでしょうか。2進数では数字は0と1しかなく、1の次は桁が繰り上がり10(イチゼロ)となります。また、2進数ではそれぞれの桁をビットと呼びます(bitの語源はbinary digitの略)。

では、2進数で

10011010

は、10進数でいくつでしょうか。これも、10進数と同じ考え方でよいのです。

2の7乗の位‥‥‥1×128	＝	128
2の6乗の位‥‥‥0× 64	＝	0
2の5乗の位‥‥‥0× 32	＝	0
2の4乗の位‥‥‥1× 16	＝	16
2の3乗の位‥‥‥1× 8	＝	8
2の2乗の位‥‥‥0× 4	＝	0
2の1乗の位‥‥‥1× 2	＝	2
2の0乗の位‥‥‥0× 1	＝	0
合計		154

はい、答えは154ですね。では逆に、154を電球8個を使って表すにはどうしたらよいでしょうか。もうおわかりですね。「0：消灯、1：点灯」として

(1, 0, 0, 1, 1, 0, 1, 0)

のような状態にすればよいですね。

もし、電球ではなくメモリに154という数字を置くにはどうすればよいでしょうか。メモリは、イメージ的には電球の代わりにコンデンサがずらっと並んだものだと思えばいいでしょう（あくまでもイメージ的にですよ）。コンデンサには、電気を蓄えた状態と放電した状態があります（これもあくまでもイメージです）。電気を蓄えた状態を1、放電した状態を0と考えれば、電球と同じ考えで154という数字を表すことができますね。このコンデンサ1個が1ビットです。

Column

1024

メモリに情報を書き込んだり、ハードディスクから情報を読み出したりするのは1バイト単位で行われます。そこでバイトのことを「基本単位」と呼ぶことがあります。ちなみに

1024バイトを1キロバイト（1KB）
1024キロバイトを1メガバイト（1MB）
1024メガバイトを1ギガバイト（1GB）
1024ギガバイトを1テラバイト（1TB）

と呼びます。

1024は2の10乗なので、2進法にとって区切りのよい数字なのです。

一方、ハードディスクの容量などでは1メガバイトを1000キロバイトで計算しているものが多いようです。また、1024単位でものを考えるとき、Kをケーと発音して区別する人もいます。

ちなみにメモリには8ビット(1バイト)ずつ、0番地、1番地、2番地、……というように住所が割り当てられています。これをメモリの**アドレス**といいます。当然1つの番地にはビットが8個並んでいます。「100番地に154という値を置く」ということは、100番地のメモリを(1, 0, 0, 1, 1, 0, 1, 0)の状態にすることです(図1-3)。

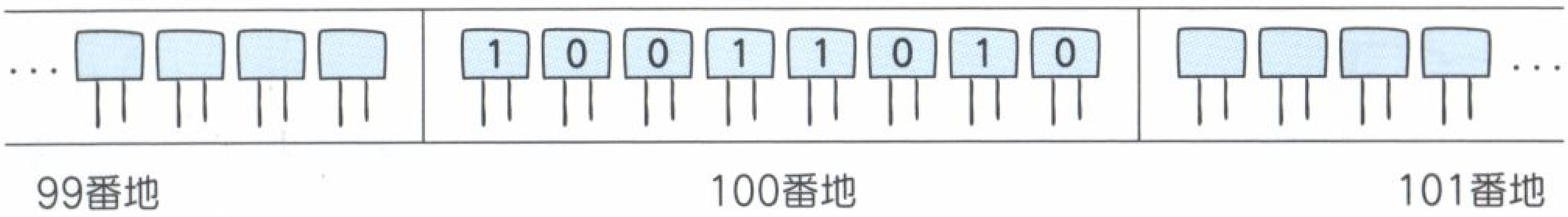


図1-3　メモリの100番地に154という値を置く

ここまでは、わりとすんなり理解できたのではないでしょうか。しかし、次のような疑問がわきませんか？

- 第1の疑問「大きな数字はどうするの？(100番地には0から255までの数値しか置けない！)」
- 第2の疑問「マイナスの数字はどうするの？」
- 第3の疑問「小数点のつく数字はどうするの？」

はい。これには、約束事を作らなくてはいけません。

大きな数値の表現

100番地に10進数の0から65535までの数値を置くとします。このとき100番地だけではビットが足りないので「101番地も使う！」と決めます。たとえば12345は2進数で表すと次のようになります。

0011 0000 0011 1001 ・・・・・(1)

このとき(0011 0000)を**上位バイト**、(0011 1001)を**下位バイト**といいます。便宜上、上位バイトをA、下位バイトをBで表すと(1)は

(AB)

となります。

ABと並ぶように、つまり上位バイトを100番地(下位番地)、下位バイトを101番地(上位番地)に置く方式を**ビッグエンディアン**といいます。これに対して、BAと並ぶように、つまり上位バイトを101番地(上位番地)、下位バイトを100番地(下位バイト)に置く方式を**リトルエンディアン**といいます(図1-4参照)。

Column

16進数

コンピュータ内部では、データは2進数で処理されていますが、2進数は桁が多く人間には認識しづらいため、プログラミングの世界では16進数も使われます。文字コードなども16進数で表現されています。2進数を下から4桁ごとに区切って置き換えると16進数になります(表1-2参照)。

表1-2　2進数、10進数、16進数対応表

2進数	10進数	16進数
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

	上位バイト	下位バイト
12345 =	0 0 1 1 0 0 0 0	0 0 1 1 1 0 0 1
	100番地	101番地
ビッグエンディアン	0 0 1 1 0 0 0 0	0 0 1 1 1 0 0 1
	100番地	101番地
リトルエンディアン	0 0 1 1 1 0 0 1	0 0 1 1 0 0 0 0

図 1-4　2バイトの場合のビッグエンディアンとリトルエンディアン

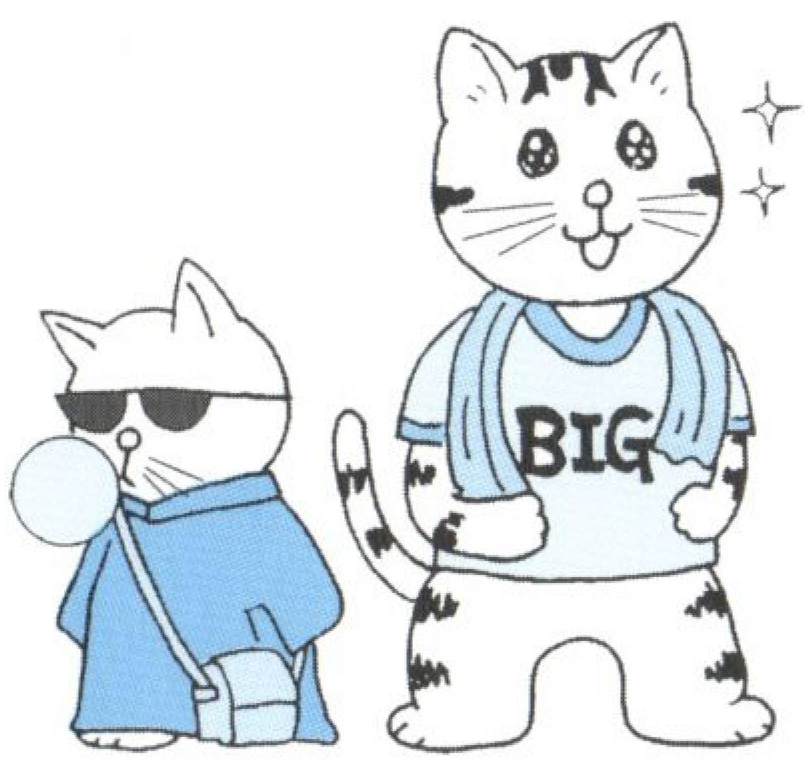
モトローラ社のCPUはビッグエンディアン、インテル社のCPUはリトルエンディアンを採用しています。

「もっと大きな数字はどうなるの？」

はい、たとえば4バイト必要な数字を扱うとします。このときは100番地から103番地までを使います。この数値が(ABCD)であったとすると、図1-5のようになります。

	100番地	101番地	102番地	103番地
ビッグエンディアン	A	B	C	D
リトルエンディアン	D	C	B	A

図 1-5　4バイトの場合のビッグエンディアンとリトルエンディアン



大人のほうが素直

紙に書いて眺めた場合、ビッグエンディアンのほうが素直ですね（「大人（BIG）のほうが素直」と覚えます）。

重要

> 大きな数値は複数のバイトで表現する。

負数の表現

さて、第２の疑問です。負の数値を取り扱うにはどうしたらよいのでしょうか。１バイトの数値を０から順番に並べてみます。

表 1-3　1 バイトの数

2 進数	10 進数
0000 0000	0
0000 0001	1
0000 0010	2
0000 0011	3
⋮	⋮
1111 1101	253
1111 1110	254
1111 1111	255

もし、10 進の –3 を数学的に２進数で表すには

–(0000 0011)

でよいのですが、コンピュータの内部では「–」のような符号をそのまま取り扱うことはできません。なんらかの工夫が必要となります。そこで、マイナスの値も扱う場合は、255 を –1、254 を –2、253 を –3、… に割り当てることにします。

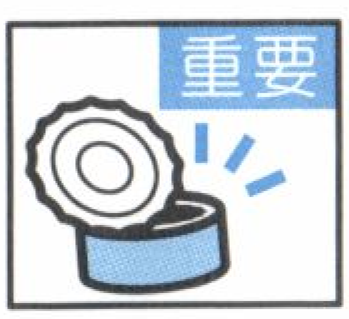
表 1-3 を書き直してみます。

表 1-4　負数を考慮した1バイトの数

2進数	10進数	マイナスを考慮した10進数
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
0000 0011	3	3
⋮	⋮	⋮
0111 1110	126	126
0111 1111	127	127
1000 0000	128	–128
1000 0001	129	–127
⋮	⋮	⋮
1111 1101	253	–3
1111 1110	254	–2
1111 1111	255	–1

マイナスを考慮しなければ1バイトで0から255までの整数を扱えますね。マイナスを考慮すると扱える整数の範囲は–128から127までとなります。

「なーーんか、わかりにくいな…」と思われるかもしれませんが、この表をよく見てみると、先頭のビットが1であるものはマイナスの数値です。この先頭のビットを**符号ビット**(sign bit)と呼びます。符号ビットが1であるものはマイナスなのです。符号ビットが0であるものは、0とプラスの数値です。



負数は符号ビットを使って表現する。

2進数の加算

では少し話題を変えて、2進数の足し算をしてみましょう。マイナスは考慮しないことにします。10進数と同じに縦書きの計算をしてみます。1＋1が出てきたらその桁は0にして、次の桁に1繰り上がります。

0101 0001	81
＋0011 1001	57
1000 1010	138

はい、よくできました。では、次はどうなるでしょうか。

1111 1111	255
＋0000 0001	1
1 0000 0000	256

数学的にはあってますね。ところが、コンピュータの世界では1バイトの計算をしていて桁があふれた場合、あふれたものは無視されます。この規則に当てはめると

1111 1111	255
＋0000 0001	1
0000 0000	0

な、なんと255＋1が0になってしまいました。(0000 0001)はマイナスを考慮してもしなくても1ですね。1に(1111 1111)を足したら0になりました。ということは、(1111 1111)は−1ということなんですね。

(0000 0010)は2です。これに、(1111 1110)を足してみましょう。

1111 1110	?
＋0000 0010	2
0000 0000	0

桁あふれを無視すると0になりましたね。ということは、(1111 1110)が−2ということなのです。

コンピュータの内部では、数は有限なので、このようなことが起こるのです。桁あふれについては、実際のプログラミングでよく起こります。そして、「何だ、こりゃーーー」という、予想外の結果をもたらします。桁あふれについての知識ができた皆さんはもう大丈夫ですね。

さて、このくどい説明で次の3つの重要なことがわかりました。

1. あるバイト数で数値を表すとき、マイナスも考えるかどうかで、表すことのできる数値の範囲が異なる。
2. 同じビットの並びでも、マイナスを考慮するかどうかで全然違う数値になってしまう。
3. 「桁あふれ（オーバーフロー）」は思いがけない結果となる。

これらは、C 言語で数値を取り扱う上で非常に重要なポイントです。

Column

ネットワークバイトオーダー

2 バイト以上の数値を取り扱うときの順番を**バイトオーダー**（byte order）といいます。メモリに数値を置くときや、データを送信するときにどのバイトから実行するかという意味です。

ところで、インターネットではさまざまなコンピュータが混在してつながっているので、バイトオーダーを統一しておかなくては混乱が起こります。そこでインターネットではビッグエンディアンが採用されています。もともとインターネットはUNIX マシンで始められました。UNIX にはモトローラ社のCPU が使われていたので、当然ビッグエンディアンが採用されたわけです。

ネットワークで採用されているバイトオーダーを、**ネットワークバイトオーダー**といいます。ネットワーク関係のプログラムを書くときには、常にこのネットワークバイトオーダーを意識しなくてはいけません。



桁あふれに注意！

実数の表現

さて、第3の疑問です。「小数はどうするの！」

これが、一番わかりにくいかもしれません。わからなくてもプログラミングにはまったく支障をきたさないので、気にしないでください。

小数を表すときに、私たちは

12.54

などと表現するのが普通ですね。小数点より右にある数字は常に1より小さいのです。小数点の位置が固定しているという意味でこのような表現方法を**固定小数点表示法**（fixed-point representation)」といいます。

しかし、科学技術計算などでは次のように表すことが多いでしょう。

1.25×10^{12}

コンピュータの世界では、これを

1.25E12

などと表現します。E は Exponential（指数）の意味です。1.25 を**仮数**、12 を**指数**、10 を**基数**と呼びます。この「1.25E12」は、

.125E13

Column

2の補数

2進数で書き表した数値で、各桁について0なら1、1なら0に書き換えたものを**1の補数**(complement on one, ones complement)といいます。また、このようにビットを書き換えることを**ビット反転**といいます。さらに「1の補数」に1を加えたものを**2の補数**(twos complement)といいます。

8桁の2進数(1バイト)で「2の補数」について考えてみましょう。(0000 0011)は10進数で3ですね。これの「2の補数」を求めてみましょう。

まず、1の補数を作ると(1111 1100)ですね。これに1を加えると(1111 1101)です。これを10進数にすると、あら不思議! −3になりました(17ページの表1-4で確認してください)。

2進数で、ある数にマイナス1をかけたものを求めるには2の補数を求めればよいのです。一般的にx進法でn桁の数Nの「xの補数」といえば、(xのn乗−N)で求めることができます。これを単に「補数」と呼ぶこともあります。

次に10進数で1桁の「4」の「10の補数」を求めてみましょう。

10の1乗−4=6

ですね。10進の1桁の数と、マイナスも考慮した数との対応表を作ってみます。

0 —— 0
1 —— 1
2 —— 2
3 —— 3
4 —— 4
5 —— −5
6 —— −4
7 —— −3
8 —— −2
9 —— −1

「6」はマイナスも考慮した数では−4です。

ある種の資格試験では必ずといっていいほど、「何タラの2の補数を求めなさい」という問題が出ます。これからその手の試験を受けようとしている人はぜひ「2の補数」について理解しておきましょう。

とか

12.5E11

などのように書き換えることができます。小数点の位置がフワフワ動き回るので**浮動小数点数**(floating point number)といいます。

これを2進数で表現するには、どうしたらよいのでしょうか。多くの場合IEEE数値規格に準じた方法で表します。C言語で扱える浮動小数点数には、単精度と倍精度があります。通常、単精度浮動小数点数は32ビット、倍精度は64ビットを使用します。2進法なので基数は常に2とわかっているので省略されます。また、ビットの先頭は符号ビットとなります。そして残りのビットで指数と仮数を表現します(図1-6)。

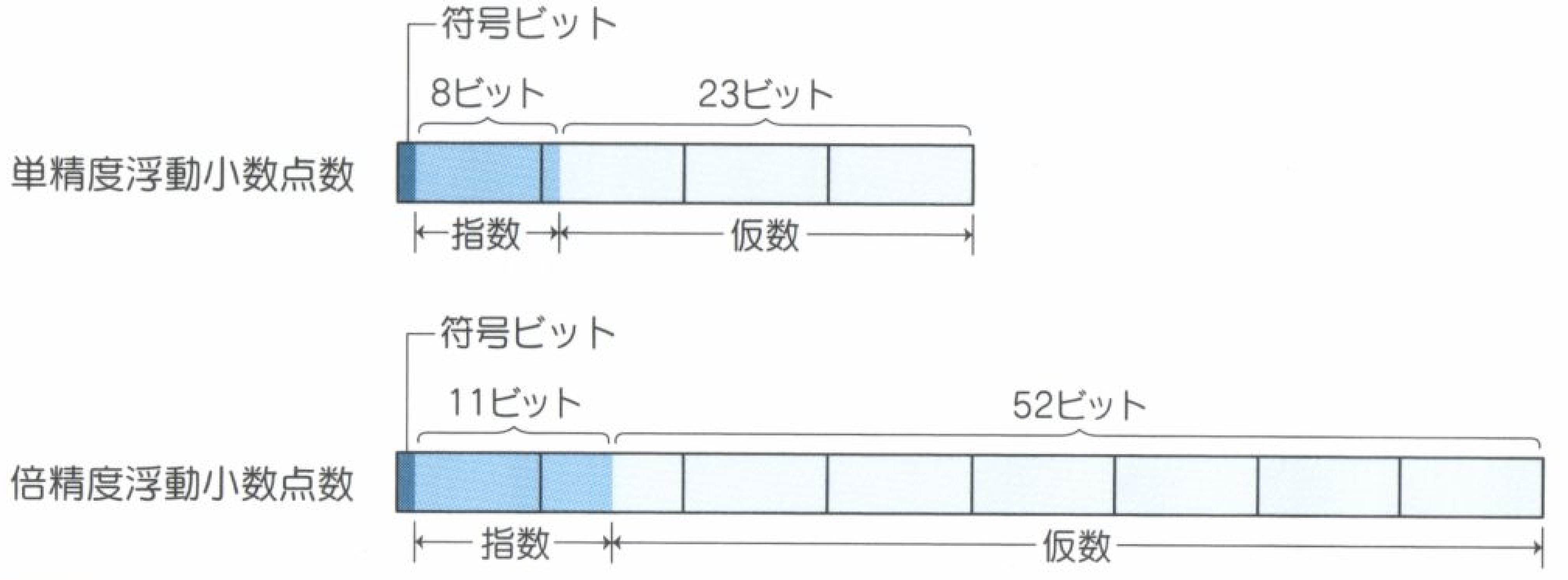


図1-6 浮動小数点数のビット表現

単精度浮動小数点数について説明します。

符号ビットはプラスなら0、マイナスなら1となります。

指数部は8ビットありますから、0から255までを表すことができます。指数がマイナスのこともあるので、127を0と考えます。128を1、129を2、130を3、… とします。すると最大値は128となります。127が0なので、126は−1、125は−2、… となり、最小値は−127となります。

要するに127を引いてやればよいのです。こういった表現法を**イクセス表現**といいます。また、この127を**バイアス**と呼びます。

では、仮数部はどうなっているかというと、2進法で1.xxxx… となるよう

に指数部を調整します。すると、先頭ビットは常に1なので、これを省略できます。このように、データを一定の規則に従って、扱いやすいように変形することを**正規化**（normalization）といいます（文字列検索のとき使う正規表現は「regular expression」です。混同しないでください）。最終的に浮動小数点数は2進法で

(SXXX XXXX) (XMMM MMM) (MMMM MMMM) (MMMM MMM)

という感じで表現されます。Sは符号ビット、Xは指数ビット、Mは仮数ビットで先頭の1が省略されたもの。なんとなくわかったでしょうか。もし今、完全に理解できなくてもまったく問題ありません。プログラムを作るうちに自然と理解できるようになります。心配せずに第2章へ進んでください。

練習問題

1. カッコ内に適当な語句や数字を入れて、文章を完成させてください。

 a. 情報量の基本単位は（　1　）である。
 b. 1バイトは（　2　）ビットである。
 c. ビットには（　3　）と1の状態がある。
 d. コンピュータの5大機能とは、（　4　）、（　5　）、（　6　）、（　7　）、（　8　）の5つの機能のことである。
 e. 10進数の11を16進数で書き表すと（　9　）になる。
 f. 2進数で書き表した数値の、各ビットの0と1を反転したものを（　10　）という。
 g. 実数を「1.25E12」のように表現したとき、1.25を（　11　）、12を（　12　）という。
 h. C言語でプログラムを作成するには、まず（　13　）ファイルという（　14　）ファイルを作成する必要がある。
 i. C言語のソースファイルを機械語に翻訳するには、（　15　）と呼ばれるソフトウェアを使用する。

※解答は付録CD-ROMに収録

第１章のまとめ

1 コンピュータには、制御・演算・記憶・入力・出力という５大機能が備わっている。

2 プログラムとはCPUに仕事の手順を命令するもの。

3 プログラムを作るには、プログラミング言語でソースファイルを記述する。

4 ソースファイルを機械語に翻訳することをコンパイルといい、生成されたファイルをオブジェクトファイルと呼ぶ。

5 オブジェクトファイルやライブラリをリンクすると、実行可能なファイルが生成される。

6 ビット(bit)とは情報量の最小単位で、２種類の状態を表現できる。

7 バイト(byte)とは８ビットの情報量で、256種類の状態を表現できる。

8 コンピュータの内部では、データは２進数で処理されている。

9 データやプログラムは、メモリ上に配置される。

10 メモリ上の位置をアドレスという。

11 大きい数値を扱う場合は、複数のバイトを用いる。

12 負数(マイナスの数)は符号ビットを用いて表現する。

13 小数の表現には、固定小数点数と浮動小数点数がある。

第2章 C言語の基本

2.1 最初のプログラム

では、実際にC言語でプログラムを作ってみましょう。どんな言語でも、一番最初に作るプログラムは「Hello World！」と画面に表示するものと相場が決まっています。「Hello World！」がイヤな人は、他の言葉でもかまいません。自分の名前でもいいでしょう。ともかく、記念すべき最初のプログラムなので慎重に考えて決めてください。では、ソースを見てみましょう。

```
/* hello.c */

#include <stdio.h>

int main()
{
    printf("Hello World!¥n");
    return 0;
}
```

このソースコードをテキストファイルに保存してコンパイルします。ソースファイルの拡張子としては、一般に「.c」が使われます。

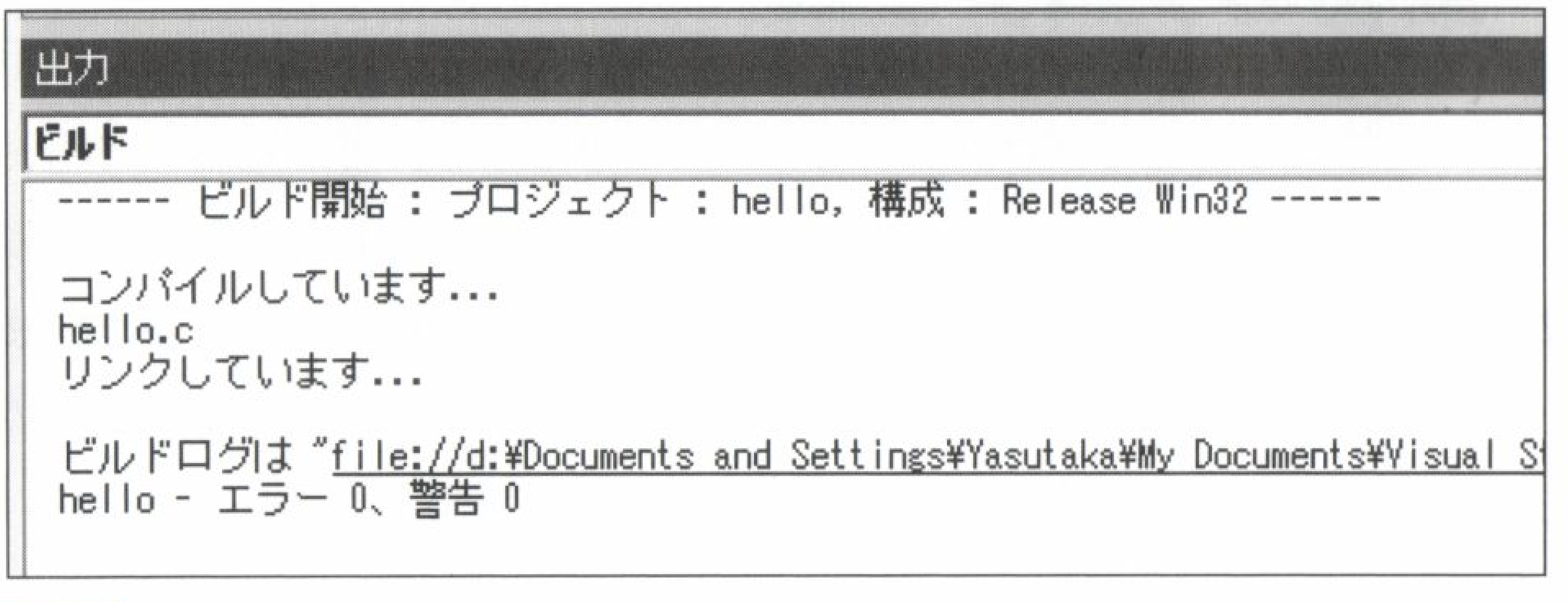
C言語の場合、大文字・小文字は区別されるので注意してください。また、ダブルクォーテーションの中身やコメント(後出)以外は、半角文字でないといけません。

続いてこのファイルをコンパイルするわけですが、その手順は開発環境によって異なります。Visual C++ Express Editionをお使いの場合は、本書の付録A(323ページ)に使用方法を掲載してあるので参考にしてください。これ以外の場合は、製品のマニュアルを参照してください。



- 大文字と小文字を区別する。
- 半角英数字で記述する(スペースも半角)。

警告もエラーも出ずに正しくコンパイルできたでしょうか。



```
出力
ビルド
------ ビルド開始: プロジェクト: hello, 構成: Release Win32 ------

コンパイルしています...
hello.c
リンクしています...

ビルドログは "file://d:¥Documents and Settings¥Yasutaka¥My Documents¥Visual S
hello - エラー 0、警告 0
```

図2-1 Visual C++ 2008の場合

さて、このプログラムを実行してみると……

```
Hello World!
```

「Hello World !」と表示されたら大成功です。では、少しだけプログラムの内容を説明しましょう。

最初の「/*」から「*/」は**コメント**(comment、remark、note)といって、プログラマの覚え書きみたいなものです。直接プログラムには関係ありません。プログラムの冒頭に限らず、どこにでもコメントを書くことができます。ここでは、ソースファイルの名前(hello.c)を覚え書きとして書いています。

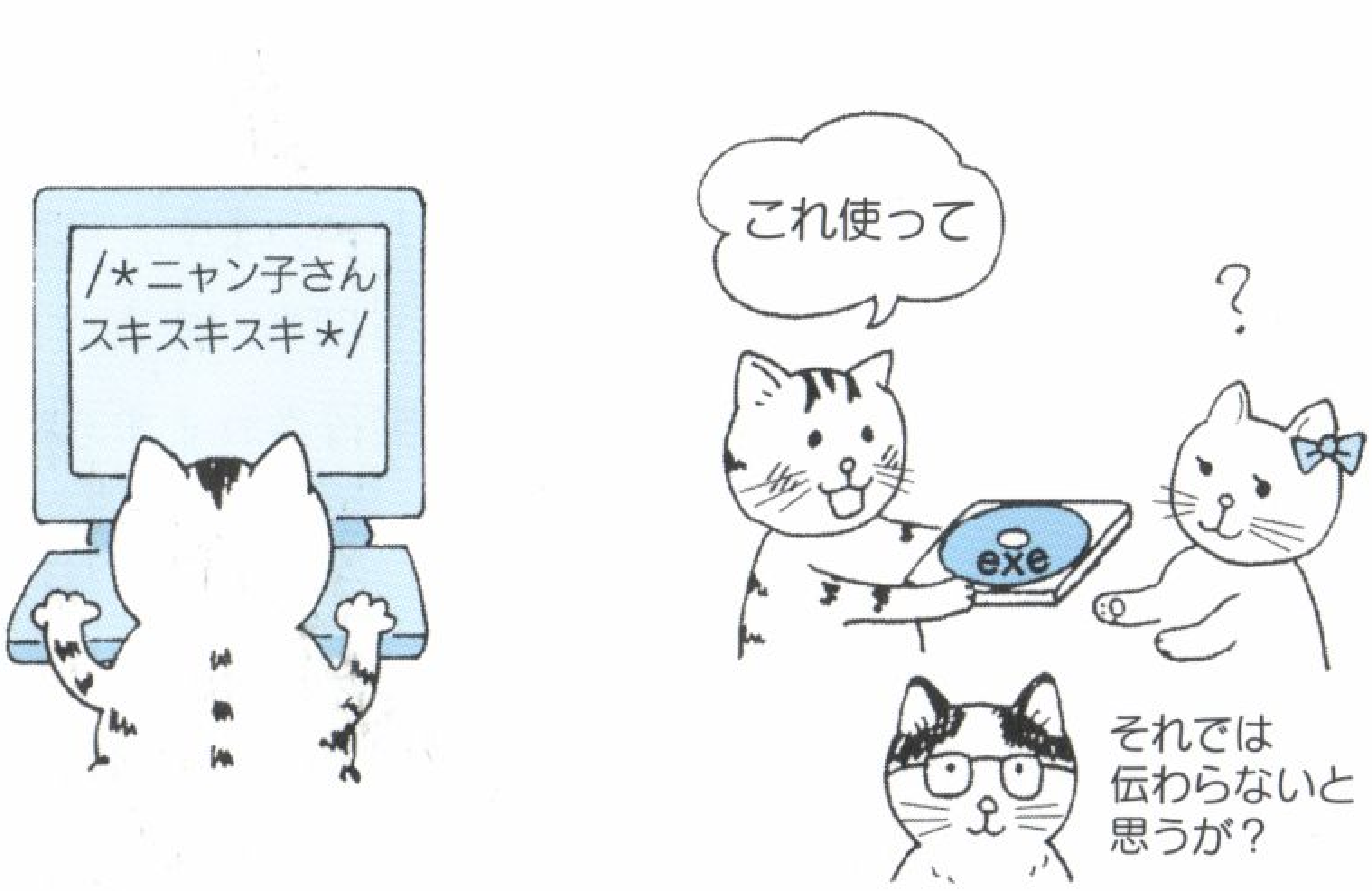
プログラムを作り始めた日時などを書くこともあります。プログラムの説明を書いておくと後で役に立ちます。ともかく、何でもいいのです。もちろんなくてもかまいません。

しかし、コメントを上手に使うかどうかで、プログラミングの上達に影響するともいわれていますので、こまめにコメントを書く習慣をつけたほうがよいでしょう。

いずれにせよ、コメントはコンパイラに無視されます。コメントの内容はできあがったexeファイルにはまったく反映されません(上司の悪口をコメントに書いてもexeだけではばれません。しかし、

「君、ちょっとソースを見せてくれないか」

といわれると、困ったことになります)。



コメントはexeに含まれない

重要

ソースコードにはコメント（覚え書き）を。

最初のコメントの次に何もない空の行があります（注：C言語では「行」という概念はないのですが、便宜上この本では「行」という言葉を使います）。C言語では、空行は無視されます。ここでは単にプログラムを見やすくするために空行を入れただけです。気にくわない人は、詰めて書いてもかまいません。

次の

```
#include <stdio.h>
```

は、いったいどういう意味でしょうか。stdio.hというファイルを取り込め、という意味です。後の章で詳しく解説しますのでここではあまり神経質になる必要はありません。

5行目の「int main()」は、int型の値を返す関数（function）を表しています。いきなり「関数」「int型」「返す」などという言葉が出てきて「あーー、なにか難しそーー」などと嘆いてはいけません。言葉は難しいのですが、たいしたことはありません。intはデータ型の1つで、単なる整数（厳密にいうとちょっと違うけど……）だと思ってください。データ型も後の章で解説します。

C言語のプログラムは極論すると関数の集合体なのです。どこを見ても関数だらけなのですぐに慣れます。関数については、後の章で詳しく解説します。

mainという関数があって、その中身が「{」から「}」の間に書かれているのです。「{」から「}」で囲まれた部分を**ブロック**（block）といいます。重要なことは、C言語のプログラムでは必ず、main（メイン）という関数がただ1つ存在するということです。プログラムはこのmain関数から実行されます（**エントリポイント**といいます）。

重要

C言語のプログラムには必ずmain関数があり、プログラムはここから開始される。

Column

コメントのいろいろ

コメントは複数行にわたってもかまいません。

```
/* コメント
   ここにもコメント
   これもコメント
   いっぱいコメント */
```

注意すべき点は「/* …… */」の中にさらに「/* …… */」を書くこはできません（コメントはネストできない）。しかし「/* …… */」の中に「/*」は書くことが可能です（コンパイラによっては警告が出ることもあります）。

```
/* abcdef /* ghijkl /* mmmm */
```

上のコメントの書き方は間違いではないのですが、紛らわしいのでやめておいたほうが無難です。

コメントをきれいに書くのが流行ったこともあります。今でも、美しいコメントを書くのを生き甲斐にしている人もいます。

```
/*****************************************
*                                        *
*   Hello.c                              *
*                                        *
*       Copyright(C) 2008 Y.Kumei        *
*                                        *
*****************************************/
```

また、C言語ではコメントに「//」を使うこともできます。「//」から行末までがコメントになります。

```
// 行末まですべてコメント
```

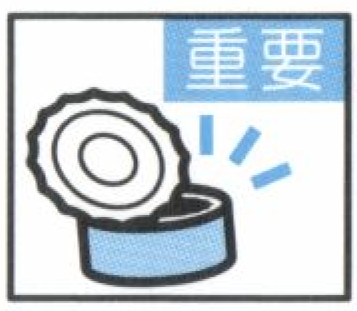
従来、C言語では「//」を用いたコメントは認められていませんでした。しかし1999年のISO規格「ISO/IEC 9899: 1999」(C99)から「//」を用いた表記も認められるようになりました。ちなみにこれはC++のコメント表記でもあります。

「main」の次にかっこがありますが、これがその前にあるmainが関数名であることを表しています。この例ではmain関数の後ろのかっこには何も書いてありませんが、一般には**引数**(argument)を書きます。これも、後から解説しますので気にしないでください。

main関数の中身を見てみましょう。このプログラムでは、中身は

```
printf("Hello World!¥n");
return 0;
```

しかありません。C言語では、1つの処理単位を**文**(statement)といいます。そしてC言語では、文の終わりには必ずセミコロン(;)をつけなくてはいけません。無意識のうちに右手の小指がセミコロンのキーに来るようになったらしめたものです。



文の最後には必ずセミコロン(;)をつける。

このサンプルのmain関数には2つの文が含まれていました。文は原則的に、最初から順番に実行されます。最初の「printf」は関数です。何をする関数かというと、標準出力(stdout)に文字を表示するのです。

「標準出力って何？」　今は「画面」と覚えておきましょう(標準出力を画面以外にすることもできます。しかし通常は画面です)。

printfは「プリント・エフ」と発音することが多いようですが、特に決まった読み方はありません(筆者は昔、これを「プリントフ」と発音していましたが、さすがにこれは少数派です)。

どうしてつづりの最後に「f」がついているかというと、これには「format付き(書式付き)」の意味が込められています。C言語では「なんとかf」という関数がけっこうたくさんありますので今後、注意して見ていてください。

```
printf("……");
```

このダブルクォーテーションの中身が表示されます。さて、ダブルクォーテ

ーションの中身の最後にある「¥n」は「改行」の意味です。

「¥」の次に来る文字は最初の「¥」と合わせて特別な意味を持ちます。これを**エスケープシーケンス**（escape sequence）とか、**拡張表記**といいます。これも、後で詳しく解説します。ここでは、「¥n」が「改行」である、ということだけ覚えておいてください。8行目の

```
return 0;
```

は、何でしょうか。厳密に説明すると難しいのですが、システムに値「0」を返しているのです。main関数はint型の値を返す関数であると最初に書きました。だから、int型の値（0）を返しているのです。これまた、関数のところで説明しますので深く考えないでください。時間の無駄です。

さて、この章でもっとも大事なことは、画面に文字を表示するプログラムを実際に作れるようになることです。hello.cの

```
printf("……¥n");
```

の"……¥n"の部分をいろいろ書き換えてみてください。そして、コンパイルして実行してみてください。

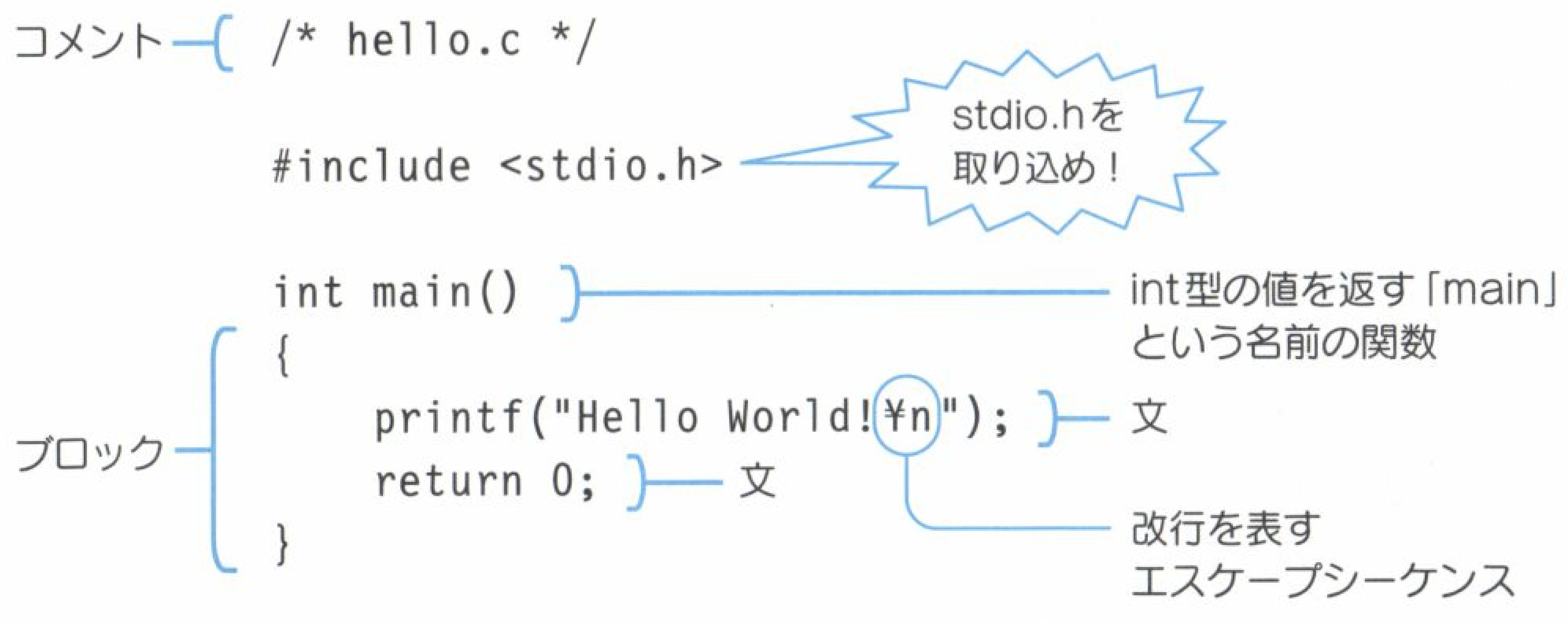


図 2-2　**hello.cの構造**

さて、最初のプログラムを次のように書き換えてみましょう。

```
#include<stdio.h>
int main(){printf("Hello World!¥n");return 0;}
```

これも、問題なくコンパイルできましたね。では、

```
#include<stdio.h>int main(){printf("Hello World!¥n");return 0;}
```

はどうでしょうか。これは、だめですね。#includeのある行は

```
#include <xxxxx>
```

で1行としなくてはなりません。では、これはどうでしょうか。

```
#include <stdio.h>

int main()
{
    printf(
        "Hello World!¥n"
    );
    return 0;
}
```

これも、大丈夫でしたね。次は、ちょっと高級です。

```
#include <stdio.h>

int main()
{
    printf(
        "Hello "
        "World!¥n"
    );
    return 0;
}
```

これはどうでしたか。大丈夫ですね。実行結果は

```
Hello
World!
```

と2行に渡って表示されるのではなく、今まで通り

```
Hello World!
```

と１行で表示される点に注意してください。ダブルクォーテーションで囲まれた部分が長くなる場合、これを分割してそれぞれをダブルクォーテーションでくくってもよいのです。

2.2 printf 関数とエスケープシーケンス

さて、前節では初めてのプログラムとして、画面に文字列を表示するプログラムを作りました。この節では、これをもう少し発展させます。画面に、

```
今日はよい天気です。
明日もよい天気でしょう。
```

と２行にわたって表示するにはどうしたらよいでしょうか。おそらく

```
/* hello02.c */

#include <stdio.h>

int main()
{
    printf("今日はよい天気です。¥n");
    printf("明日もよい天気でしょう。¥n");

    return 0;
}
```

というプログラムを書いたでしょう。しかし、次のようにも書けます。

```
001: #include <stdio.h>
002:
003: int main()
004: {
```

```
    printf("今日はよい天気です。¥n明日もよい天気でしょう。¥n");

    return 0;
}
```

「¥n」が改行を表しているので上のようなプログラムも書けるのですね。「¥」に文字が続く場合、これをエスケープシーケンスといって特別な意味を持つことはすでに31ページで解説しました。エスケープシーケンスには他にどのようなものがあるのでしょうか。

表2-1　エスケープシーケンス

エスケープシーケンス	意味
¥n	改行(New line) 現在の印字位置を次の行の先頭位置に移動する
¥t	タブ(horizontal Tab) 次の水平タブ位置に移動する
¥b	バックスペース(Backspace) 現在の行で前に移動する。先頭にある場合は不定。
¥r	キャリッジリターン(carriage Return) 現在行の最初の位置に移動
¥f	ページフィード(Form feed) 次の論理ページの最初に移動
¥'	シングルクォーテーション(single quotation mark) 一重引用符
¥"	ダブルクォーテーション(double quotation mark) 二重引用符
¥0	ヌル文字(null) 空文字
¥¥	円記号(¥)
¥?	クエスチョンマーク(?)
¥a	ベル音(Alert) ベル音を鳴らす。印字位置は不変
¥xhh	16進拡張(heXadecimal) 16進でhhのコードを持つ文字

表 2-1 エスケープシーケンス（続き）

エスケープシーケンス	意味
¥ooo	8進拡張（octal） 8進で ooo のコードを持つ文字

では、エスケープシーケンスのタブ（¥t）を利用した簡単なプログラムを見てみましょう。

```
/* escape.c */

#include <stdio.h>

int main()
{
    printf("商品名¥t定価¥t個数¥t小計¥n");
    printf("------------------------------¥n");
    printf("テレビ¥t20,000¥t1¥t20,000¥n");
    printf("ラジオ¥t5,000¥t2¥t10,000¥n");
    printf("本¥t1,000¥t12¥t12,000¥n");
    printf("------------------------------¥n");
    printf("合計¥t¥t¥t42,000¥n");
    return 0;
}
```

実行結果は次のようになります。

```
商品名  定価     個数     小計
------------------------------
テレビ  20,000  1        20,000
ラジオ  5,000   2        10,000
本      1,000   12       12,000
------------------------------
合計                     42,000
```

では、次にバックスペース(¥b)はどのように働くか実験してみましょう。

```
/* backspace.c */

#include <stdio.h>

int main()
{
    printf("abcdefg");
    printf("¥b");
    printf("hijklmn");
    printf("¥bopqrstuvwxyz¥n");

    return 0;
}
```

実行結果がどのようになるか予想してみましょう。

```
abcdefhijklmopqrstuvwxyz
```

アルファベットのｇとｎが抜けていますね。

それでは、次のような文字列を出力したいときはどのようなプログラムになるでしょうか。

```
"readme.txt"ファイルはC:¥doc¥doc01にあります。
```

```
/* yen.c */

#include <stdio.h>

int main()
{
    printf("¥"readme.txt¥"ファイルは C:¥¥doc¥¥doc01にあります。¥n");

    return 0;
}
```

ダブルクォーテーションそのものを出力したいときは「¥"」になります。また、円記号そのものを出力したいときは「¥¥」になります。うっかり間違えやすいので注意しましょう。



ダブルクォーテーションを表示するには「¥"」。
円記号(¥)を表示するには「¥¥」。

エスケープシーケンスにはまだまだいろいろな使い方がありますが、さしあたってはここで紹介したサンプルプログラム程度を知っていれば、困ることはないでしょう。

これまでに紹介したサンプルプログラムを単に眺めるだけではなく、必ず実際に自分で入力して、プログラムを作る手順をマスターしてください。

練習問題

1. カッコ内に適当な語句を記入して、C言語のプログラムを完成させてください。

```
#（　a　） <stdio.h>

int （　b　）()
{
    printf("Hello world!¥n");
    （　c　） 0;
}
```

2. printf関数を使って、画面に次のように表示するプログラムを作ってください（改行に注意）。

```
ほ乳類は、恒温動物である。
人間は、ほ乳類である。
故に、人間は恒温動物である。
```

3. printf関数を使って、画面に次のように表示するプログラムを作ってください（「¥」の表示に注意）。

```
このノートは¥105です。
```

※解答は付録CD-ROMに収録

第2章のまとめ

1 C言語はmain関数から開始される。

2 文の最後には「;」（セミコロン）をつける。

3 ソースコードにはコメントをつけることができる。

4 文字を表示するにはprintf関数を使う。

5 いろいろなエスケープシーケンス（拡張表記）がある。

第3章

変数とデータ型

今までのプログラムは、単に文字列を表示するだけで何かもの足りないですね。なぜでしょうか。それは変数を使っていないプログラムなので、プログラムらしくないのです。この章では、

- 変数
- データ型

について考えます。

3.1 変数とは

変数 (variable) とは、値 (データ) を入れる (記憶する) 入れ物です。単に値を入れるだけではなく、これに対して掛けたり、割ったり、足したりといった演算をすることが可能です。次の数式を見てください (プログラムではなく単なる算数として)。

```
a = 10        ……(1)
b = 20        ……(2)

c = a + b     ……(3)
```

この場合、cの値は何でしょうか。

そう、30ですね。このときaやbやcを変数といいます。この雰囲気がC言語のプログラムに当てはまります。

(1)や(2)では変数a、bに数値を代入していますね。(3)では、代入された値からcの値を計算しています。

これを、ちょっと数字を書き換えて、

```
a = 2         ……(1)
b = 10        ……(2)

c = a + b     ……(3)
```

としたらどうでしょうか。aやbに入れる数字が変わりました。cの値は今度は12です。しかし、(3)そのものの式は同じですね。つまり、(3)の式が1つあれば、aやbの値をいろいろ変えることにより、いろいろな足し算ができるということです。変数を使うメリットがなんとなく理解できたでしょうか。



変数は、値を入れる入れ物。

ここで、プログラムの話に戻ります。C言語では原則的に文の終わりにセミコロンをつけるので、

```
a = 10;
b = 20;

c = a + b;
```

というような書き方をします。数学では

```
x = 10
```

と書くと、「x は 10 に等しい」という意味ですね。しかし、C 言語の世界では

```
a = 10;
```

と書くと、「変数 a に 10 を代入しなさい」という意味になります。したがって数学では

```
10 = x
```

という書き方も成り立ちますが、プログラムでは

```
10 = a;
```

はエラーとなります。あくまでも「=」の左にある変数に、右の値を代入するという意味だからです。また、数学では

```
x + y = 10
```

という数式は成り立ちますがプログラムではエラーとなります。あくまでも「=」の左側にあるのは 1 つの変数です。

では、これはどうでしょうか。

```
a = 1;
a = 10;
```

最初に変数 a に 1 を代入しました。次に変数 a に 10 を代入しました。「あれ、じゃあ a の値はどうなったのかな？」 はい、変数には最後に代入した値しか残りません。最初の 1 はきれいさっぱり忘れ去られます。これは、非常に大事な性質です。



- 「=」は左にある変数に、右の値を代入する。
- 変数には最後に代入した値だけが残る。



また、

```
a = a + 1;
```

というような書き方もよく見かけます。「=」は等しいという意味ではなく、代入するという意味です。つまり「a + 1 の値を a に代入しなさい」という意味です。このような変数の概念があるかどうかが、低水準言語（アセンブリ言語）と高水準言語（C 言語など）の分類の判断基準の 1 つとなっています。

3.2 データ型

さて、データの型（type）にはおおざっぱに

- 整数型
- 浮動小数点型
- 文字型

があります。整数型はそのサイズにより、**short**、**int**、**long** 型に分類されます。大きさは

short ≦ int ≦ long

となります。どの型が何ビットであるかはコンパイラにより異なります。昔(俗にいう 16 ビット時代)は、short は 8 ビット、int は 16 ビット、long は 32 ビットであることが多かったです。しかし、現在のコンパイラでは short は 16 ビット、int は 32 ビット、long も 32 ビットであることが多いようです。したがって int も long も大きさが同じなので、long はほとんど使われなくなりました。

int 型が 32 ビットであるとすれば、ここに格納できる整数の範囲は

$$-2^{31} \sim +2^{31} - 1$$

となります。つまり、−21 億 4748 万 3648 から 21 億 4748 万 3647 の範囲となるわけです。日常的な整数値は int 型で間に合いますね。

浮動小数点型には **float**、**double**、**long double** 型があります。サイズはコンパイラにより異なりますが float 型が 32 ビット、double 型が 64 ビット、long double 型が 64 ビットであることが多いようです。「じゃあ、具体的に float 型とか double 型は、いくつからいくつまでの範囲の実数を扱えるのだ?」という質問が来そうですね。この答えは、整数型のようにはっきりいくつからいくつといえません。float 型はおよそ 3.4E−38 から 3.4E+38 の範囲です。double 型はおよそ 1.7E−308 から 1.7E+308 の範囲です(3.4E−38 とは 3.4×10^{-38} という意味です)。

文字型は **char** 型です。C 言語では、文字と文字列は違うものなので注意してください。文字や文字列については第 8 章で詳しく解説します。文字とは、'a'、'b' などのように、シングルクォーテーションで囲まれた 1 文字です。char 型にはちょっとびっくりするような性質がありますが、これも第 8 章でゆっくり解説します。char 型はどのコンパイラでも 8 ビットです。



データ型には主に、整数型(int、short、long)と浮動小数点型(float、double、long double)と文字型(char)がある。

3.3 変数の宣言

変数を使うには、どういったデータを格納するのかブロックの最初で**宣言**(declaration)しなくてはいけません。たとえば、int型のデータを格納する変数aを宣言するには、次のように記述します。

```
int a;
```

変数名は、英字かアンダースコア(_)で始まる英数字列となります。変数の名前の長さの制限はコンパイラによって決まっています。

```
int nNo;
double _kion01;
```

ただし、C言語にはあらかじめ特定の目的のために使うことが決められている**予約語**(reserved word、key word)があり、これを変数として使うことはできません。しかし、変数の一部に含むことは可能です(forは予約語なのでこれを変数として扱うことはできませんが、afor, for1, a_forなどは変数として使用可能です)。



変数を使うときは最初に宣言すること

表 3-1　予約語

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

また、次のように、同じ型の変数を一度に宣言することも可能です。

```
int a, b, c;
double d, e, f;
```

Column　宣言は「文」か？

変数の宣言は、

```
int a;
```

のように書きました。セミコロンで終わっていますね。これは「文」なのでしょうか。ある言語では、これは「宣言文」という文の一種として扱われます。しかし、C言語において文は実行する動作を指定するものなので、通常は文として扱いません。

また、C言語では{ }で囲まれた部分をブロックということはすでに解説しました。このブロックに書かれている文を**複文**（compound statement）ということがあります（たとえ1つの文しかなくても）。「あれ？！変数の宣言は文ではないといっておいて、ちょっと矛盾しているぞ！！！」

```
int main()
{
    int a;
    a = 10;
    printf("a = %d¥n", a);
    return 0;
}
```

確かに{ }の中に文以外のもの（変数の宣言）が含まれていますね。C言語では複文に宣言を含めることができるのです（ちょっとこじつけ的ですが…）。

変数に値を入れることを**代入**(assignment)といいます。変数の宣言と同時に、変数に代入することも可能です。これを変数の**初期化**(initialization)といいます。

```
int a = 100, b, c = 50;
```



・ 変数を使うには、データ型と変数名を宣言する。
・ 変数に値を入れることを代入という。

3.4 変数の値の表示

さて、変数を宣言したり、変数に数値を代入できるようになっても、これを表示できなければ面白くありません。

```
printf("文字列");
```

とすれば、ダブルクォーテーションでくくられた文字列が表示されました。この文字列中に「%○」(○はd, f, cなど)が存在すると、その部分に、ダブルクォーテーションの後ろにある文字や数値を埋め込むことができます。整数のときは**%d**、浮動小数の場合は**%f**、文字の場合は**%c**を指定します。

「何だかよくわからんなーーーー!」

では、例を示しましょう。

```
printf("1年は%d日です¥n", 365);
```

とすれば、

```
1年は365日です
```

と、表示されます。%dの部分が365に置き換わったのです。この置き換えは、複数行うことができます。

```
printf("%d日は %d年です¥n", 365, 1);
```

最初の%dは365に、次の%dは1に置き換えられて、「365日は1年です」と表示されます。さて、これと前節でやった変数を組み合わせて次のようなこともできます。

```
/* format01.c */

#include <stdio.h>

int main()
{
    int a, b, c;

    a = 10;
    b = 15;

    c = a + b;

    printf("%dたす%dは%dです¥n", a, b, c);

    return 0;
}
```

変数aに10を代入

変数bに15を代入

変数cに(a+b)の値を代入

これは、次のように書き換えることも可能です。

```
/* format02.c */

#include <stdio.h>

int main()
{
    int a, b;

    a = 10;
    b = 15;
```

```c

    printf("%dたす%dは%dです¥n", a, b, a + b);

    return 0;
}
```

実行結果はいずれも次のようになります。

```
10たす15は25です
```

もっと例題を示すとわかりやすいかもしれません。

```c
/* format03.c */

#include <stdio.h>

int main()
{
    double a = 0.5, b = 10.5;
    int c = 215, d;
    char e = 'A';


    printf("%f + %f = %f¥n", a, b, a + b);
    d = c + 11;
    printf("cの値は%dでこれに11を加えると%dとなります¥n", c, d);
    printf("eには¥"%c¥"が代入されています¥n", e);

    return 0;
}
```

実行結果は次のようになります。

```
0.500000 + 10.500000 = 11.000000
cの値は215でこれに11を加えると226となります
eには"A"が代入されています
```


さて、ここでちょっと不満があると思います。aやbの値は小数点以下1桁なのに表示では小数点以下6桁も表示されています。これは、次のようにすれば希望がかないます。

%fとか%dなどは**書式指定フィールド**と呼ばれています。書式指定フィールドは次のように書き表すことになっています。

%[flags][width].[precision]{typeのプレフィックス}type

%とtype以外は全部省略可能です。%dとか%fとかいうのは目一杯省略しているのです。

こういう記述を見ると、「こりゃ、だめだ！わからん！」といって投げ出してしまう人はいませんか。難しそうに見えますが、どうということはありません。flagsとかwidthなどと英語らしき単語が並んでいるので難しく見えるのです。単なる記号だと思って、まずは表3-2をざっと眺めてみてください。

さて、これだけの説明では何のことだかわかりませんね。

%5.2f

とあったら、表示する桁数は全部で（小数点も含めて）5桁、小数点以下は2桁です。1.26は

1.26

と表示されます。全部で5桁なのでまだ1桁余っていますね。余った分は1桁分空白で表示されています。%の次に－がない場合は右詰めになるのです。

%-5.2f

とすると

1.26

と表示されます。

表 3-2　書式指定フィールド

フィールド	指定値	意味
flags	–	変換される引数を左詰めにします。指定しないと右詰めになります。
	+	出力する値が符号付きの場合「+」または「–」の符号をつけます。指定しないとマイナスのときのみ「–」がつきます。
	0（ゼロ）	最小幅になるまで0を追加します。出力値がマイナスと0の場合は0は追加されません。省略すると0は追加されません。
width	数値	文字幅を指定します。
precision	数値	精度を指定します。つまり小数点以下の桁数を指定します。
type のプレフィックス	h	short int に適応させます。
	l	long 修飾です。
	L	long double 修飾です。
type	d，i	10進（Decimal）整数を指定します。
	o（小文字のオー）	符号なし8進数（Octal）を指定します。
	u	符号なし（unsigned）10進整数を指定します。
	x	符号なし16進（heXadecimal）整数で、a，b，c，d，e，fを使います。
	X	符号なし16進（heXadecimal）整数で、A，B，C，D，E，Fを使います。
	f	浮動小数点（Floating-point）を指定します。
	e，E	指数（Exponent）変換を指定します。±○.○○e○○または±○.○○E○○。
	c	文字（Character）変換を指定します。
	s	文字列（String）変換を指定します。
	p	ポインタ（Pointer）変換を指定します。

%05.2f

では、どうなるでしょうか。全部で５桁、小数点以下は２桁、頭があいたら０で埋める、という意味ですね。

01.26

と表示されます。

重要

書式指定フィールドで、表示書式を指定できる。

では、サンプルのプログラムを示します。

```
/* format04.c */

#include <stdio.h>

int main()
{
    double pai = 3.14159265358979;
    int mon = 2;
    float flt = 1.2f;

    printf("%5.2hf¥n", flt);
    printf("%-5.2f¥n", flt);
    printf("%05.2f¥n", flt);
    printf("%+08.2f¥n", flt);
    printf("%-08.2f¥n", flt);

    printf("%d¥n", mon = 3);

    printf("%e¥n", pai);

    printf("%08.2f¥n", pai);
    printf("%05d¥n", mon);
```

```
    printf("円周率は%fです¥n", pai);
    printf("円周率は%lfです¥n", pai);
    printf("もう少し詳しい値は%10.8lfです¥n", pai);
    printf("もう少し詳しい値は%15.13lfです¥n", pai);
    printf("もう少し詳しい値は%20.18fです¥n", pai);
    printf("円周率は%010.2f¥n", pai);
    printf("円周率は%-10.2f¥n", pai);
    printf("円周率は%10.2f¥n", pai);
    return 0;
}
```

実行結果は次のようになります。ソースプログラムとじっくり対比してみてください。

```
 1.20
1.20
01.20
+0001.20
1.20
3
3.141593e+000
00003.14
00003
円周率は3.141593です
円周率は3.141593です
もう少し詳しい値は3.14159265です
もう少し詳しい値は3.1415926535898です
もう少し詳しい値は3.141592653589790000です
円周率は0000003.14
円周率は3.14
円周率は      3.14
```

Column
型の明示的な表現

float 型の変数に値を代入するとき、

```
float a;
a = 1.5;
```

とすると、コンパイラが警告を発します。これは、単に「1.5」と書くと double 型の 1.5 であると解釈されるからです。float 型の変数よりも大きなサイズの double 型の数値を代入しようとしているとコンパイラが判断して「おい、おい、危ないぞ！」と警告を発しているのです。

1.5 が float 型の 1.5 であるということを示すには、1.5 の後ろに「f」または「F」をつけます。

```
a = 1.5f;
```

こうすれば、コンパイラは文句をいいません。また、数値が long double 型であることを示すには数値の後ろに「l」または「L」をつけます。f, F, l, L のようなものを**浮動小数点接尾語**（floating suffix）」といいます。

整数に関しても同様の接尾語があります。これを**整数接尾語**（integer suffix）といいます。これには unsigned を表す u, U と long を表す l, L があります。

また、C 言語では、16 進数の数値を表すとき

```
0x45
```

のように先頭に 0x（数字のゼロと英字のエックス）をつけます。また、8 進数の場合は先頭に 0（ゼロ）をつけます。次のように書くと、8 進数の 125 ということになります。

```
0125
```

練習問題

1. カッコ内に適当な語句を入れて、文章を完成させてください。

   a. 変数を使うためには、まず変数の（　1　）を行わなくてはならない。

   b. データ型には大きく分けて、（　2　）型、（　3　）小数点型、文字型がある。

c. 浮動小数点型には主に、float型、(　4　)型、long double型がある。
d. 整数型には主に、short型、(　5　)型、long型がある。
e. 文字型には、(　6　)型がある。

2. カッコ内に適当な語句を入れて、「円周率は3.14です」と表示するプログラムを完成させてください。

```
#include < (  a  ) >

int main()
{
    double pai = 3.14159;
    printf("円周率は(  b  )です¥n", pai);
    (  c  )  0;
}
```

※解答は付録CD-ROMに収録

第3章のまとめ

1 変数とは値を入れる入れ物である。

2 「=」は左の変数に右の値を代入する。

3 データ型には大きく分けて、整数型、浮動小数点型、文字型がある。

4 整数型には主に、short型、int型、long型がある。

5 浮動小数点型には主に、float型、double型、long double型がある。

6 文字型には、char型がある。

7 変数を使うためには、まず変数を宣言する。

8 printf関数で変数の値を表示する際は、書式指定フィールドを指定することで、いろいろな書式で表示できる。

第4章 式と演算子

4.1 式と演算子

式(expression)とは何でしょうか。これを厳密に定義するとおそらくほとんどの人は、この本を投げ出してしまうでしょう(それは困る!!!)。おおざっぱにいって、式とは

i = 20

のような数式であると考えてください。これは「変数iに20を代入する」という意味ですね。C言語では、決して「iと20は等しい」という意味ではないことを再確認してください。「=」を**代入演算子**(assignment operator)といいます。**演算子**は、特定の操作機能を有した記号です。演算子は別名「作用素」とも呼ばれます。

C言語には演算子が実にたくさん存在しています(いかん、「たくさんある」などと書くと読者のヤル気がうせてしまう)。いや、C言語は演算子が豊富なのでプログラミングが楽です(ものは言いようです)。

いずれにせよ、心配するには及びません。基本的な演算子は数学（というより算数）のそれとほとんど同じです。

足し算は「+」で、これはすでに何回も出てきました。

引き算は「-」です。

かけ算は「*」です。

割り算は「/」です。これは分数をイメージするとわかりやすいですね。5÷3を5/3と書くのは日常的にもよく行われます。

さて、演算子の作用対象を**オペランド**（operand、演算項目）といいます。iと20がこれにあたります。

これは納得できましたね。しかし、この場合、iや20も式と考えることができます。

「えーーっ！？ 何かへんだなーー」

このように式のもっとも基本となる要素を**一次式**（primary expression）と呼びます。式の最後にセミコロンをつけると文になります。式のみからなる文を**式文**（expression statement）といいます。ま、名前くらいは覚えておいてください。

式は**評価**（evaluation）されて値を持ちます。「20」という式は評価されて20という値を持ちます。この値が変数（iという式）に代入されます。この結果、「i = 20」という式全体の値が20となります。「えーーっ、ちょっと変だな」と思われるかもしれませんが、これでよいのです。

```
3 + 2
```

も式です。この式の値が5であることは納得しますね。3も式で、評価されて3という値を持ちます。2も評価されて2という値を持ちます。そして、式全体が5という値を持ちます。

前章で、

```
printf("%d + %d = %d¥n", a, b, a + b);
```

というのをやりましたね。3番目の%dのところには、式「a+b」が評価されその値が表示されるのです。

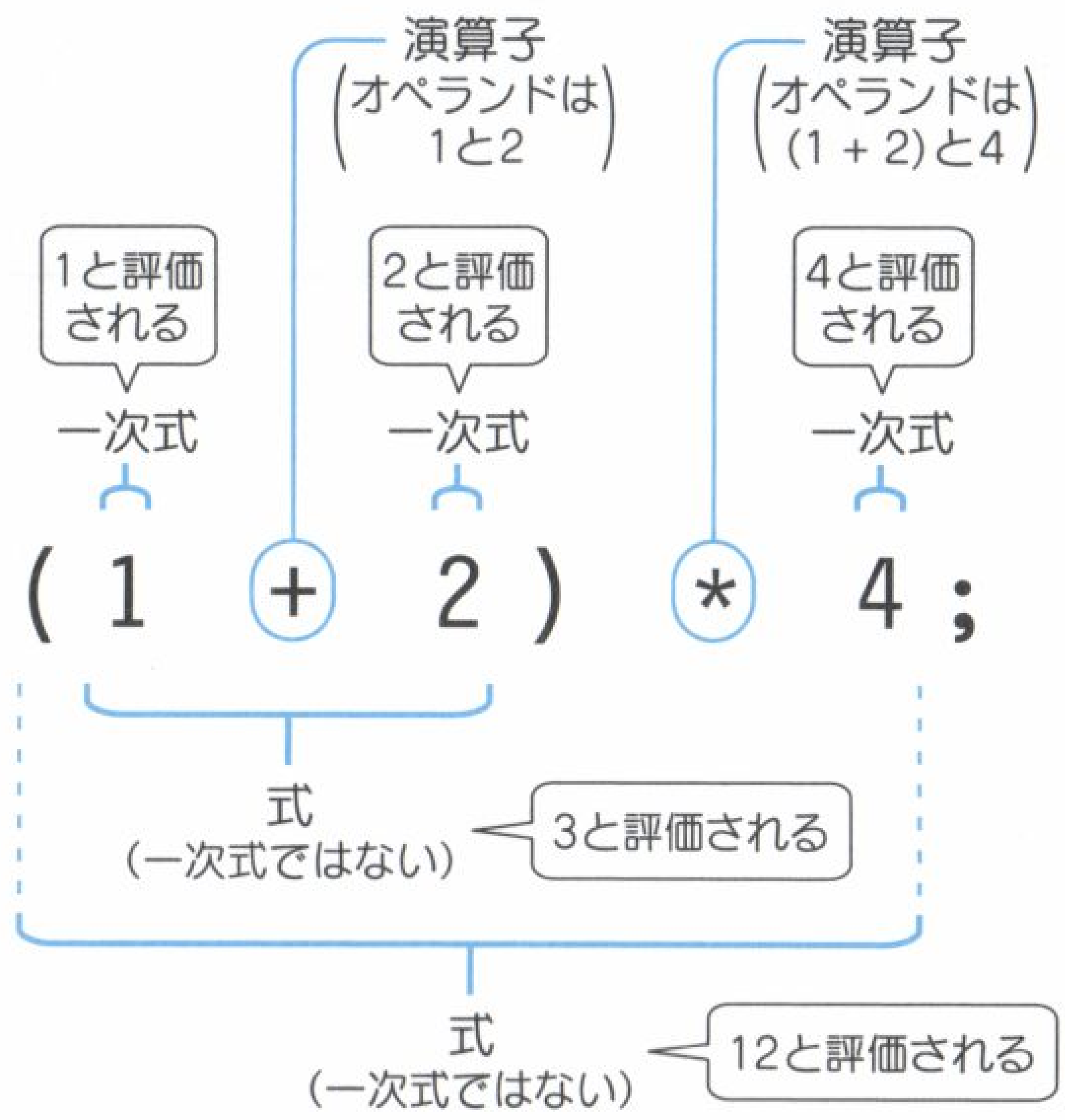
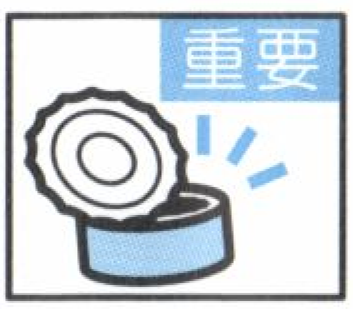


図 4-1　**式と演算子**



式は、評価されて値を持つ。

では、i = 20 という式の値が本当に 20 であるかどうか試してみましょう。

```
// op01.c

#include <stdio.h>

int main()
{
    int i;

    printf("「i = 20」の式の値は%dです¥n", i = 20);
    return 0;
}
```

これを実行すると

```
「i = 20」の式の値は20です
```

と表示されましたね。「へぇーー、そんなもんかいな」などといっていないで、実際に自分でプログラムを書いて実行してみてください。

では、次はどのように考えればよいのでしょうか。

```
i = 2;
j = i + 3;
```

「j はiに3を足すのだな…。で、iは2なのでjには5が入っているのか…」

常識的には、= よりも + を先に実行しますね。それでよいのです。これを**演算子の優先順位**(precedence) といいます。そして、その優先順位はたいていは常識的に納得のいくものなのです。

この例では、

1. まず + が優先されるので、i + 3 が評価される。
2. そして、式の値は5となる。
3. この5がjに代入される。

となります。

Column

式の評価順序

j = 2 + 3についてもう少し考えてみましょう。

2 + 3が先に評価されることはわかりました。では、2と3のどちらが先に評価されるのでしょうか。順番に2から評価されるのでしょうか。

実はC言語では定義されていません。コンパイラによって異なるのです。

ま、この場合、結果としてはどちらでもよいのですが ……。

重要

演算子には優先順位が決められている。

4.2 データ型と計算

さて、これで計算はかなりできるようになりました。しかし、この「計算」が曲者(くせもの)です。同じ型どうしの計算をして、結果として他の型になってしまうことがよくあるのです。

たとえばint型の10とint型の3で割り算をしてみます。期待される結果はdouble型の3.3333… ですね。

また、10 + 1.5のように違う型の計算はどうなるのでしょうか。あれこれ考えるよりプログラムを作って確かめてみましょう。

```
// op02.c

#include <stdio.h>

int main()
{
    int a = 10, b = 3;
    double c;

    c = a / b;

    printf("%d ÷ %d = %f¥n", a, b, c);
    return 0;
}
```

10÷3は3.3333…
になるはずだが…

これは、同じ型どうしの演算をして、結果は別の型になることを期待して作ったプログラムです。a，bはint型で、得られる結果cをdouble型にしたいと思います。printf関数の中でもcに対応する変換指定は%fで間違いないですね。

では、実行してみましょう。

```
10 ÷ 3 = 3.000000
```

これは、おかしいですね。どうしたことでしょう。実はC言語では、同じ型どうしの演算を行うとその結果の値は元の型と同じになります。intどうしの割り算なので結果もint型となります。このとき切り捨てが起こり3となります。これを%fで表して3.000000となるのです。

「じゃ、異なる型の計算はどうなるの？」

はい、試してみましょう。

```
/* op03.c */

#include <stdio.h>

int main()
{
    int a;
    double b;

    a = 10;
    b = 2.35;

    printf("%d + %f = %f¥n", a, b, a + b);

    return 0;
}
```

intとdoubleの足し算

これを実行してみると、次のようになります。

```
10 + 2.350000 = 12.350000
```

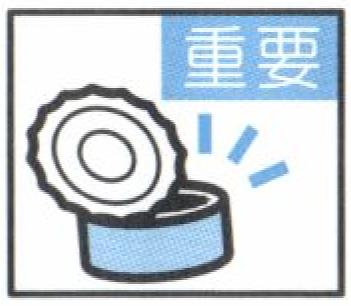
これは、いいようですね。C言語では異なる型どうしの計算を行うときは、大きい型に合わせられるのです。これを**格上げ**(promote)といいます。

では、代入はどうなるのでしょうか。

```
a = b;
```

の代入を行うとき、bがどのような型であってもaの型になってしまいます(これは重要です。蛍光ペンを持っている方はマークをつけてください。持っていない方は、すぐに文房具店に行って買ってきてください)。代入により暗黙のうちに型が変換されてしまうのです。

たとえば、int型の変数aに1.9を代入すると、結果は1.9がint型に変換されます。このとき、丸めが起こって2となるのか、切り捨てが起こって1になるかは、コンパイラの仕様により異なるので注意が必要です。



- 同じデータ型どうしの計算では、値は元の型と同じになる。
- 異なるデータ型どうしの計算では、値は大きいデータ型に合わせられる。
- 代入を行うと、値は代入先の型に合わせられる。

4.3 各種演算子

では、C言語で定義されている、いろいろな演算子を紹介しましょう。C言語には表4-1の演算子があります。この表を暗記する必要はまったくありません。必要が生じたらこの表を調べてください。

表4-1 演算子

優先順位	記号	意味	結合規則
1	()	関数呼び出し。printf(...), main()等	→
	[]	配列	→
	->	構造体メンバ参照	→
	.	構造体メンバ参照	→
	++	後置増分	→
	--	後置減分	→
2	++	前置増分	←
	--	前置減分	←
	sizeof	記憶量	←
	&	アドレス	←
	*	間接参照	←
	+	正符号	←
	-	負符号	←
	~	補数(ビット反転)	←
	!	否定	←
3	()	キャスト	←
4	*	乗算	→
	/	除算	→
	%	剰余	→
5	+	加算	→
	-	減算	→
6	<<	ビット左シフト	→
	>>	ビット右シフト	→

表 4-1 演算子(続き)

優先順位	記号	意味	結合規則
7	<	左不等号(大きい)	→
	<=	等価左不等号(以上)	→
	>	右不等号(小さい)	→
	>=	等価右不等号(以下)	→
8	==	等価	→
	!=	非等価	→
9	&	ビット積	→
10	^	ビット差	→
11	|	ビット和	→
12	&&	論理積	→
13	||	論理和	→
14	?:	条件。a?b:cのように使う	←
15	=	代入	←
	+=	加算代入	←
	-=	減算代入	←
	*=	乗算代入	←
	/=	除算代入	←
	%=	剰余代入	←
	<<=	左シフト代入	←
	>>=	右シフト代入	←
	&=	ビット積代入	←
	^=	ビット差代入	←
	|=	ビット和代入	←
16	,	コンマ演算子	→

演算子は優先順位の高いものから評価されます。

```
a + b * c
```

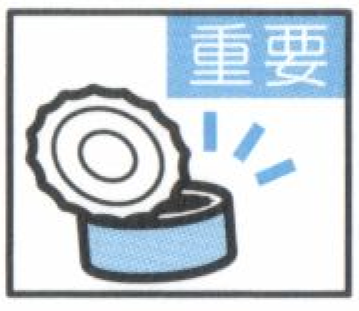
では、*のほうが優先順位が高いのでまず、b * cが評価されます。b * cの式の値がDであるとすれば

```
a + D
```

ということになります。では、同じ優先順位の演算子が複数あった場合はどうでしょうか。

```
d = a / b * c
```

「/」も「*」も優先順位は等しいですね。この場合、表 4-1 の**結合規則**(associativity) のところをみます。「→」となっていますね。この矢印の方向にそって評価されます。つまり、a / b が先に評価されます。



演算子には優先順位と結合規則がある。

演算子を複数使用するときは演算子の優先順位や結合規則をあれこれ考えてプログラムするよりも、数学と同じ考えで、先に計算してほしいところを ( ) でくくっておくと間違いが生じません。

```
/* op04.c */

#include <stdio.h>

int main()
{
    double shincho = 1.72, taiju = 65.2;

    printf("BMIは体重(kg)を身長(m)の2乗で割ったものです¥n");
    printf("これが25を超えると肥満です¥n");
    printf("たとえば身長%4.2fmで体重%4.2fkgの人では¥n",
            shincho, taiju);
    printf("BMI = %4.2f ÷ (%4.2f x %4.2f)で計算します¥n",
            taiju, shincho, shincho);
    printf("その結果は%4.1fでした¥n",
            taiju / (shincho * shincho));

    return 0;
}
```

Column　剰余演算子「%」の使い方

剰余演算子（modulus operator）「%」は比較的よく使います。これは、割り算の余りを求める演算子です。

「こんなものいったいどういうときに使うんだ！？」

実は、これがけっこう使うんです。ゲームを作るときによく乱数（ランダムな値）が必要となります。たとえば、じゃんけんゲームを作るとします。コンピュータが何を出すかは、まったくでたらめであるとします。このとき０から２までの乱数（0，1，2）を発生させて０ならグー、１ならチョキ、２ならパーと決めておきます。これで、コンピュータの出す手を決められますね。

しかし、普通Ｃ言語の関数で乱数を発生させると０からRAND_MAX（32767）までの整数が得られます。これじゃ使えないですね。そこで、得られた乱数をxとすると（x % 3）を計算するのです。ある整数を３で割ったときの余りは０か１か２のいずれかですね。



実行結果は次のようになります。

```
BMIは体重(kg)を身長(m)の2乗で割ったものです
これが25を超えると肥満です
たとえば身長1.72mで体重65.20kgの人では
BMI = 65.20 ÷ (1.72 x 1.72)で計算します
その結果は22.0でした
```

4.4 インクリメント・デクリメント演算子

数学では出てこない演算子に ++（インクリメント演算子）と --（デクリメント演算子）があります。

インクリメント演算子（increment operator）は値を 1 増やします。**デクリメント演算子**（decrement operator）は値を 1 減らします。また、これらの演算子はオペランドの後につく（後置）か先につく（前置）かで働きが変わってきます。

```
a++
```

と書くと、この式が評価された後に a の値を 1 増やします。

```
++a
```

と書くと、評価前に a の値が 1 増やされます。

同様に a-- は評価後に a の値を 1 減らします。--a は評価前に値を 1 減らします。

```
/* inc01.c */

#include <stdio.h>

int main()
{
    int a = 10;
    int b = 10;

    printf("a = %d¥n", a++);
    printf("a = %d¥n", a);

    printf("b = %d¥n", ++b);
    printf("b = %d¥n", b);
    return 0;
}
```

a を評価した後に a の値を 1 増やす

b の値を 1 増やしてから b を評価する

このプログラムを実行した結果を予測してみてください。

10行目でa++が評価されますが、このときは10のままです。これが、まず表示されます。評価が終わるとaの値が1増やされ11となります。それで、11行目が実行されると「a = 11」と表示されるわけです。また、13行目では、++bが評価される前にbの値が1増やされ11となります。したがって「b = 11」と表示されます。


```
a = 10
a = 11
b = 11
b = 11
```

さて、インクリメント（デクリメント）演算子は今までの知識で書き換えることができます。

```
/* inc02.c */

#include <stdio.h>

int main()
{
    int a, b;

    /* この書き方に注意！ */
    /* 結合規則によりb = 10が先に評価され、この式の値が10となる */
    /* a = (b = 10) であり、aに10が代入される */
    a = b = 10;

    printf("a = %d¥n", ++a);
    b = b + 1;
    printf("b = %d¥n", b);

    return 0;
}
```

重要

- 「a++」は、aが評価されてからインクリメントされる。
- 「++a」は、インクリメントされてから評価される。

4.5 sizeof演算子

見た目は演算子のように見えませんが**sizeof演算子**(sizeof operator)というものがあります。これには、2つの書き方があります。

- sizeof 式
- sizeof(型の名前)

これらによって、式や型のサイズ(バイト)を調べることができます。もちろん式にかっこをつけて

```
sizeof(式)
```

としても間違いではありません(むしろかっこをつけたほうが間違いが少なくなる)。しかし、「sizeof 型の名前」は認められません(コンパイルエラーとなります)。

さて、ここでちょっとした注意があります。sizeof演算子の返す型は、コンパイラによってint型であったり、unsigned int型であったりとまちまちです。そこで、sizeof演算子の返す型は**size_t型**と決められました。

```
/* sizeof01.c */

#include <stdio.h>

int main()
{
    char c = 'A';
```

```
    short s = 50;
    int i = -100;
    unsigned int ui = 128;
    float f = 0.5f;
    double d = 2568.2;
    long double ld= 1.258E-25;
    size_t sz;

    sz = sizeof c;
    printf("size of char = %d バイト¥n", sz);

    sz = sizeof s;
    printf("size of short = %d バイト¥n", sz);

    sz = sizeof i;
    printf("size of int = %d バイト¥n", sz);

    sz = sizeof ui;
    printf("size of unsigned int = %d バイト¥n", sz);

    sz = sizeof f;
    printf("size of float = %d バイト¥n", sz);

    sz = sizeof d;
    printf("size of double = %d バイト¥n", sz);

    sz = sizeof ld;
    printf("size of long double = %d バイト¥n", sz);

    sz = sizeof(size_t);
    printf("size of size_t = %d バイト¥n", sz);

    return 0;
}
```

各変数のサイズを調べる

実行結果は次のようになります。

```
size of char = 1 バイト
size of short = 2 バイト
size of int = 4 バイト
size of unsigned int = 4 バイト
size of float = 4 バイト
size of double = 8 バイト
size of long double = 8 バイト
size of size_t = 4 バイト
```

4.6 不等号演算子

>(左辺は右辺より大きい)、**<**(左辺は右辺より小さい)、**>=**(左辺は右辺より大きいまたは等しい)、**<=**(左辺は右辺より小さいまたは等しい)を**不等号演算子**(inequality operator)といいます。これも、数学の不等式とイメージ的には同じです。たとえば次のことがいえます。

- 「5 > 3」は正しい(真、true)
- 「5 < 3」は正しくない(偽、false)

不等号演算では、式が真である場合、式の値は1となります。偽である場合は0となります。

```
/* sl01.c */

#include <stdio.h>

int main()
{
    int a;
    a = 5;

```

```
    /* 正しい */
    printf(" a < 10は「%d」です¥n", a < 10);
    /* 正しくない */
    printf(" a > 10は「%d」です¥n", a > 10);

    return 0;
}
```

実行結果は次のようになります。

```
a < 10は「1」です
a > 10は「0」です
```

重要

- 不等号演算では、式が真の場合、式の値は１になる。
- 式が偽の場合、式の値は０になる。

4.7 等価演算子と非等価演算子

「==」は右辺と左辺が等しいときに真になります。これを**等価演算子**（equality operator）といいます。一方「!=」は**非等価演算子**（unequality operator）といい、右辺と左辺が等しくないときに真となります。式の値としては真のとき１で、偽のとき０となります。

これらは、次章で出てくるif文では非常によく使われる演算子なのでぜひとも覚えておいてください。サンプルを見てみましょう。

Column

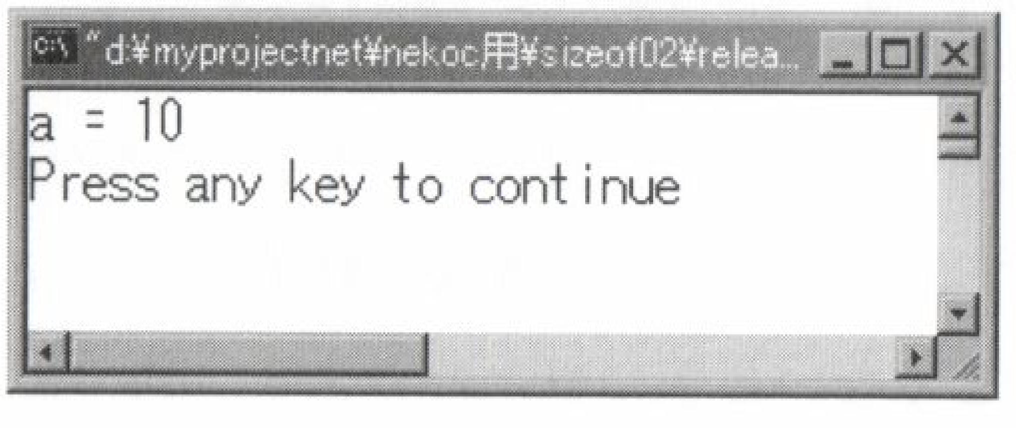
sizeof の評価

さて、次のようなプログラムの場合はどのような実行結果となるでしょうか。

```
001: /* sizeof02.c */
002:
003: #include <stdio.h>
004:
005: int main()
006: {
007:     int a = 10;
008:     size_t s;
009:
010:     s = sizeof ++a;
011:
012:     printf("a = %d¥n", a);
013:
014:     return 0;
015: }
```

「簡単、簡単！ まず10行目のsizeofのところで、aが評価される前に1増やされてaは11になる。次に12行目で画面にはa = 11と表示される。」

正しいようにも思えますが、本当にa = 11と表示されるか実験してみましょう。



あら、不思議！ どうしてa = 10なのでしょうか。実は

```
sizeof 式;
```

としても式は評価されないのです。したがって上のプログラムで「sizeof ++a;」としても「++a」が評価されないので、aの値は10のままなのです。

```
/* eq01.c */

#include <stdio.h>

int main()
{
    int a = 10;

    printf("aの値は%dです¥n", a);
    printf("a == 10の値は%dです¥n", a == 10);
    printf("a == 20の値は%dです¥n", a == 20);
    printf("a != 10の値は%dです¥n", a != 10);
    printf("a != 20の値は%dです¥n", a != 20);

    return 0;
}
```

実行結果は次のようになります。

```
aの値は10です
a == 10の値は1です
a == 20の値は0です
a != 10の値は0です
a != 20の値は1です
```



- 「==」は、右辺と左辺が等しいときに真(1)になる。
- 「!=」は、右辺と左辺が等しくないときに真(1)になる。

4.8 代入演算子

代入演算子はすでに何回も出てきました。「=」は単に値を代入するだけなので**単純代入演算子**(simple assignment operator)とも呼ばれます。さて、突然ですが復習です。

```
a = a + 6;
```

という文はどういう意味ですか？ はい、変数aの値に6を加えて、その値をaに代入するのでしたね。これと同じことを次のように書くこともできます。

```
a += 6;
```

つまり、

```
A = A + B;
```

は

```
A += B;
```

と書くことができるのです。「+=」を**加算代入演算子**(addition assignment operator)と呼びます。同様に、

- 「−=」：減算代入演算子(subtraction assignment operator)
- 「*=」：乗算代入演算子(multiplication assignment operator)
- 「/=」：除算代入演算子(division assignment operator)
- 「%=」：剰余代入演算子(modulus assignment operator)

というお仲間がいらっしゃいます。使い方は想像がつきますね。それぞれの演算結果を左辺に代入します。これらを**複合代入演算子**(compound assignment operator)といいます。では、次のサンプルプログラムを見てください。

```
001: /* as01.c */
002:
003: #include <stdio.h>
004:
005: int main()
006: {
007:     int a1, a2, a3, a4;
008:
009:     a1 = 5;
010:     a1 = a1 + 2;
011:     printf("a1 = %d¥n", a1);
012:
013:     a2 = 5;
014:     a2 += 2;
015:     printf("a2 = %d¥n", a2);
016:
017:     a3 = 5;
018:     a3 = a3++ + 2;
019:     printf("a3 = %d¥n", a3);
020:
021:     a4 = 5;
022:     printf("a4 = %d¥n", a4++ + 2);
023:
024:     return 0;
025: }
```

15行目までは簡単ですね。では、17行目以降はどうでしょうか。17行目で変数a3に5が代入されました。18行目の左辺を見てみましょう。++と+の2つの演算子があります。優先順位は++のほうが強いですね。a3++はa3が評価されてからa3に1が加えられます。この段階ではまだa3の値は5です。次に2を加えます。左辺の式の値は7となりました。

次にこの値をa3に代入します。a3は最初の評価が終わっているので1が加わります。結果としてa3は8となります。

なんだか狐につままれたみたいな話ですね。(a3++ + 2)の段階では、この式の値は7なのです。これをa3に代入したときにa3は8となるのです。

本当にこのようになっているかを示したのが、21 行目からです。(a4++ ＋ 2) の値を表示すればはっきりしますね。では、実行結果を見てみましょう。

```
a1 = 7
a2 = 7
a3 = 8
a4 = 7
```

はい、ごらんの通り、a3 のみが 8 となっています。何がいいたいかというと

A = A op B; (op は +, −, *, /, %などの演算子)

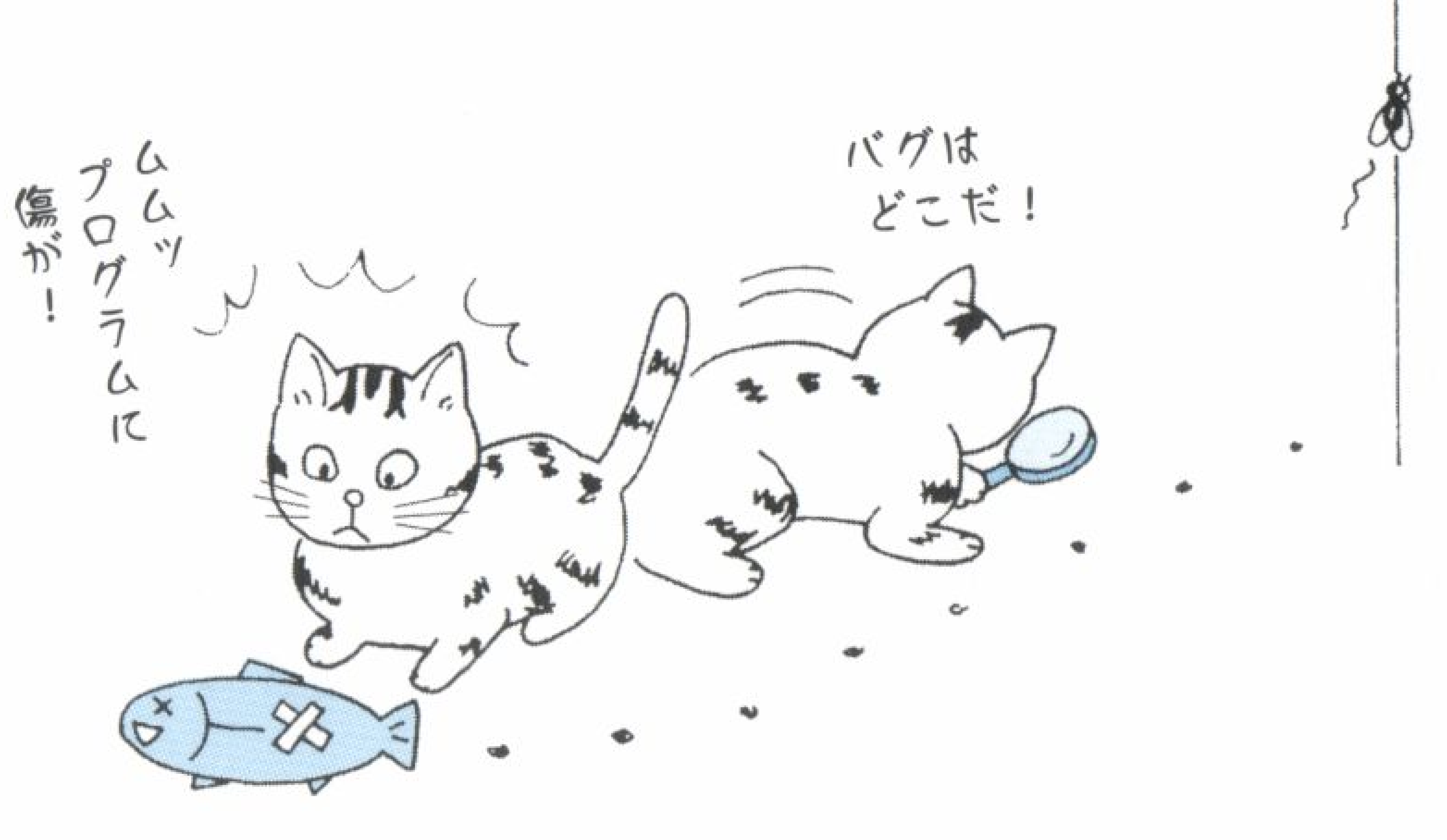
という書き方をすると、A の評価が 2 回行われる、ということです。一方、

A op= B; (op= は +=, −=, *=, /=, %= などの演算子)

という書き方では A は一度しか評価されません。

「あーー、ヤダ、ヤダ！！ C 言語って屁理屈ばっか！！」

すいません。こんな変な例は実際のプログラミングではめったに出てきません。でも、こういうことも知っておくと、プログラムが思い通りに動かないときに、バグ探しのヒントになるかもしれません。



4.9 キャスト演算子

皆さんは骨折したときにどうしますか？（おいおい、いきなりなんだ！？）病院に行ってギプスを巻いてもらいますね。ところで「ギプス（gips）」はオランダ語なのです。英語ではキャスト（cast）といいます。cast には、「鋳造する」とか「ある形にする」という意味があります。要するに無理矢理型にはめる、というような意味です。骨が曲がってつかないよう、型にはめるのでギプスのことをキャストというのですね。

実は、C 言語には**キャスト演算子**というものがあって、次のように使います。

(型名 X)式

これで、式の型を型名 X にすることができます。式にギプスを巻いて無理矢理別の型にしてしまうわけです。どんなメリットがあるかというと、次のサンプルを見てください。

```
/* cast01.c */

#include <stdio.h>

int main()
{
    int a = 10, b = 4;
    double c;

    c = a / b;

    printf("%d ÷ %d = %f…(1)¥n", a, b, a / b);
    printf("%d ÷ %d = %f…(2)¥n", a, b, c);
    printf("%d ÷ %d = %f…(3)¥n", a, b, (double)a / b);

    return 0;
}
```

int 型の変数を double 型に型キャストする

実行結果は次のようになります。

```
10 ÷ 4 = 0.000000…(1)
10 ÷ 4 = 2.000000…(2)
10 ÷ 4 = 2.500000…(3)
```

(1)は、(a / b)はint型なのに%fで表そうとしているので、問題外です。

(2)は、(a / b)はint型で値もint型です。したがって切り捨てが起こり2となります。これをdouble型の変数に代入しています。「a = b;」では、bがどのような型であってもaの型になる、という規則がありました(61ページ参照)。したがってint型の2はdouble型の2.000000となったのですね。

(3)は、int型の10が、型キャストで無理矢理double型に変換されました。型の違うものどうしの演算では、より大きな型に合わせられましたね。したがってdouble型どうしの割り算になるのです。結果として正しい値が表示されました。

Column

Windowsプログラミングにおける型キャスト

WindowsのGUIプログラム(ま、普通のWindowsのプログラムのことです)をC言語で書くときは、Win32 API(Win32 Application Programming Interface)を利用します。Win32 APIとは、Windowsがアプリケーションに提供する関数群です。これを利用するとき、やたらと型キャストが必要になってきます。

実はWin32 APIでは、星の数ほどデータ型が定義されています。これは、プログラマが誤ったプログラムを書いてしまうのを防ぐためです。

その一方では、型が多すぎるのでキャストして使わざるを得ない場面も多々あるのです。一方を立てれば片方が立たず、といったところでしょうか。

4.10 ビット演算子

ビット単位の演算を行うものを**ビット演算子**(bitwise operator)といいます。**ビット論理演算子**(bitwise logical operator)ともいいます。どんなものがあるかというと、代表的なものとして表4-2のものがあります。

表4-2 代表的なビット演算子

演算子	意味
&	ビット積演算子。ビット単位で比較してともに1なら1
|	ビット和演算子。どちらかが1なら1
^	ビット差演算子。互いに異なれば1

まずはサンプルプログラムを見てみましょう。

```
/* bit01.c */

#include <stdio.h>

int main()
{
    unsigned short a, b;

    a = 1;
    b = 3;

    printf("a & b = %d¥n", a & b);
    printf("a | b = %d¥n", a | b);
    printf("a ^ b = %d¥n", a ^ b);

    return 0;
}
```

ここでは、short 型を２バイトとします。このとき、

１を２進法で表すと　0000 0000 0000 0001
３を２進法で表すと　0000 0000 0000 0011

となります。これらの２つの数に対してビット演算を行うと、図 4-2 のような結果となります。

ビット積

ともに1なら1、どちらかが0なら0

```
  0000 0000 0000 0001    (1)
& 0000 0000 0000 0011    (3)
-------------------------------
  0000 0000 0000 0001    (1)
```

ビット和

どちらかが1なら1

```
  0000 0000 0000 0001    (1)
| 0000 0000 0000 0011    (3)
-------------------------------
  0000 0000 0000 0011    (3)
```

ビット差

異なれば1、同じであれば0

```
  0000 0000 0000 0001    (1)
^ 0000 0000 0000 0011    (3)
-------------------------------
  0000 0000 0000 0010    (2)
```

図 4-2　基本的なビット演算

では、プログラムを実行してみましょう。

```
a & b = 1
a | b = 3
a ^ b = 2
```

はい、予想通りでしたね。

ビット演算の利用方法

「ビット演算なんてどんなとき使うの？」そういった疑問が出てくると思いました。これが意外とよく使われるのです。次のサンプルを見てください。

```
/* bit02.c */

#define CAT     1    /*    CATを1と定義 */
#define DOG     2    /*    DOGを2と定義 */
#define RAT     4    /*    RATを4と定義 */
#define RABBIT  8    /* RABBITを8と定義 */

#include <stdio.h>

int main()
{
    int a = CAT | DOG;                  /* Aさんのペット */
    int b = RAT;                        /* Bさんのペット */
    int c = CAT | DOG | RAT | RABBIT;   /* Cさんのペット */
    int d = DOG | RAT;                  /* Dさんのペット */

    printf("Aさんのペット(猫=%d, 犬=%d, ネズミ=%d, ウサギ=%d)¥n",
      (a & CAT)!=0, (a & DOG)!=0, (a & RAT)!=0, (a & RABBIT)!=0);
    printf("Bさんのペット(猫=%d, 犬=%d, ネズミ=%d, ウサギ=%d)¥n",
      (b & CAT)!=0, (b & DOG)!=0, (b & RAT)!=0, (b & RABBIT)!=0);
    printf("Cさんのペット(猫=%d, 犬=%d, ネズミ=%d, ウサギ=%d)¥n",
      (c & CAT)!=0, (c & DOG)!=0, (c & RAT)!=0, (c & RABBIT)!=0);
    printf("Dさんのペット(猫=%d, 犬=%d, ネズミ=%d, ウサギ=%d)¥n",
      (d & CAT)!=0, (d & DOG)!=0, (d & RAT)!=0, (d & RABBIT)!=0);

    return 0;
}
```

a, b, c, dはAさん、Bさん、Cさん、Dさんの飼っているペットを表しています。飼っているペットを「|」演算子でつないでいます。ここで、Aさんがネズミを飼っているかどうかを調べるときは(a & RAT)の値を調べます。飼っていなければ0、飼っていれば0以外の値となります。その他の動物につ

いても同じように調べることができます。

どうしてこんなことができるのでしょうか。それには、各ペットに割り当てられている数字にヒントがあります。たとえばプログラム冒頭部分の「#define CAT 1」で、CATに1を割り当てています（#defineについては11.7節を参照してください）。これらの数字をよく見ると、CAT，DOG，RAT，RABBITの値が2倍、2倍となっていることに気づきます。これを2進法で表してみましょう（便宜上、8ビット分しか表示していません）。

CAT	00000001
DOG	00000010
RAT	00000100
RABBIT	00001000

各ペットのビットで1となっているのは1つのみです。さて、それぞれのペットのビットが1になっているかどうかは「ペットの値 & 調べたい値」でわかります。つまり、調べたい値のそのペットのビットが1になっていれば、0以外の値になります。たとえば（CAT | DOG）の値は（00000011）です。これにCAT（00000001）が含まれているかどうかを調べるには

(00000001) & (00000011)

の値を調べます。この演算結果は0以外の値となります。では、プログラムの実行結果を見てみましょう。

```
Aさんのペット(猫=1, 犬=1, ネズミ=0, ウサギ=0)
Bさんのペット(猫=0, 犬=0, ネズミ=1, ウサギ=0)
Cさんのペット(猫=1, 犬=1, ネズミ=1, ウサギ=1)
Dさんのペット(猫=0, 犬=1, ネズミ=1, ウサギ=0)
```

補数演算子

この他に「~」という演算子(**補数演算子**)があります。これは、各ビットを反転させます。

```
/* bit03.c */

#include <stdio.h>

int main()
{
    int a = 10;

    printf("a  = %08X¥n~a = %08X¥n", a, ~a);

    return 0;
}
```

実行結果は次のようになります。

```
a  = 0000000A
~a = FFFFFFF5
```

10は32ビットで表すと

(0000 0000 0000 0000 0000 0000 0000 1010) ……(1)

となり、これを16進に直すと0x0000000A。(1)のビットを反転すると

(1111 1111 1111 1111 1111 1111 1111 0101)

となり、これを16進で表すと0xFFFFFFF5ですね。

シフト演算子

シフト演算子というものもあります。これは、ビットを右または左にずらす演算子です。かけ算、割り算を高速で処理することができます。

「>>」は、ビットを右にずらします。図4-3を見てください。(1000 0000)のビットを1つ右にずらす(シフトする)と(0100 0000)となります。左にできた隙間は0で埋め、右にはみ出たビットは無視します。これは、値が2分の1になりました。さらにビットを1つ右にシフトすると(0010 0000)となり、またまた値が2分の1になりました。

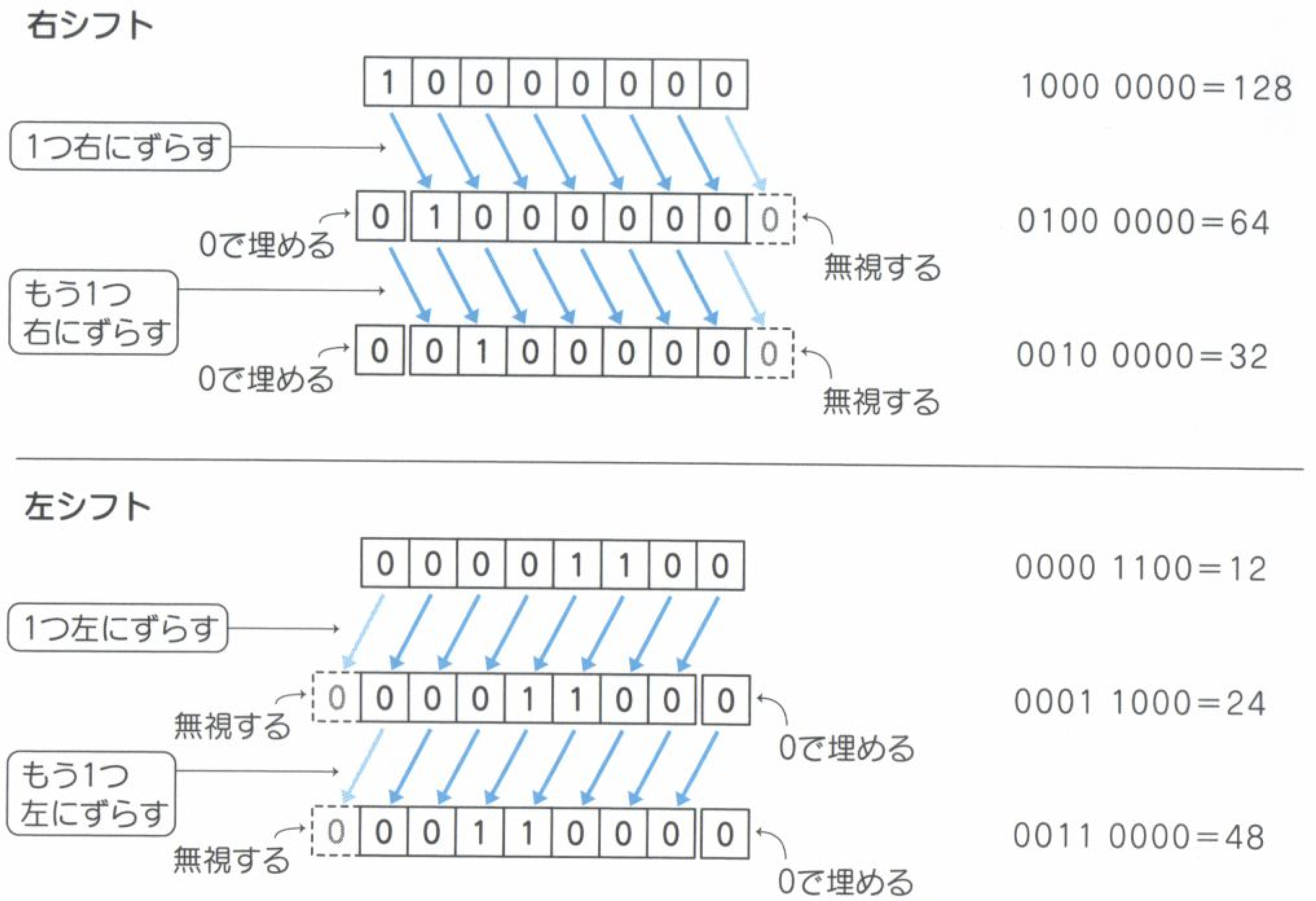


図4-3 シフト演算

```
E >> n
```

と書くと、Eをnビット右にシフトさせます。「>>」を**右シフト演算子**(right-shift operator)といいます。逆に

```
E << n
```

は、Ｅをｎビット左にシフトさせます。「<<」を**左シフト演算子**(left-shift operator)といいます。

さて、Ｅの値がマイナスの場合、シフト演算の結果はコンパイラにより異なります。

```
/* shift01.c */

#include <stdio.h>

int main()
{
    short a = 12, b = 100, c = -50;

    printf("%dを左に1ビットシフト --- %d¥n", a, a << 1);
    printf("さらに1ビット左シフト --- %d¥n", a << 2);
    printf("さらに1ビット左シフト --- %d¥n", a << 3);
    printf("さらに1ビット左シフト --- %d¥n", a << 4);
    printf("¥n");
    printf("%dを右に1ビットシフト --- %d¥n", b, b >> 1);
    printf("さらに1ビット右シフト ---- %d¥n", b >> 2);
    printf("さらに1ビット右シフト ---- %d¥n", b >> 3);
    printf("さらに1ビット右シフト ---- %d¥n", b >> 4);
    printf("(処理系依存)¥n");
    printf("%dを左に1ビットシフト --- %d¥n", c, c << 1);
    printf("%dを右に1ビットシフト --- %d¥n", c, c >> 1);

    return 0;
}
```

```
12を左に1ビットシフト --- 24
さらに1ビット左シフト --- 48
さらに1ビット左シフト --- 96
さらに1ビット左シフト --- 192

100を右に1ビットシフト --- 50
さらに1ビット右シフト ---- 25
さらに1ビット右シフト ---- 12
さらに1ビット右シフト ---- 6
(処理系依存)
-50を左に1ビットシフト --- -100
-50を右に1ビットシフト --- -25
```

4.11 条件演算子

変わり種の演算子として「?:」というのがあります。これは、**条件演算子**(conditional operator)と呼ばれるもので、次のように使います。

op1 ? op2 : op3

op1を評価してこれが真(0以外)であれば、op2が評価されます。偽(0)であればop3が評価されます。これじゃ、いまいち意味がわかりませんね。例題を見ればすぐにわかります。

```
/* conditional01.c */

#include <stdio.h>

int main()
{

    (3 > 5) ? printf("3は5より大きい¥n") :
```

```
            printf("3は5より大きくない¥n");

    return 0;
}
```

(3 > 5)は偽なので「:」の後ろにある「printf("3は5より大きくない¥n")」が評価され、実行結果は次のようになります。

```
3は5より大きくない
```

ちなみに、この条件演算子のようにオペランドを3つ使う演算子のことを**3項演算子**(ternary operator)といいます。オペランドを2つとる演算子のことを**2項演算子**(binary operator)」、オペランドを1つ取る演算子のことを**単項演算子**(unary operator)といいます。

練習問題

1. カッコ内に適当な語句を入れて、文章を完成させてください。

a. C言語でint型変数aに10を代入するには、(　1　);と書く。
b. 「=」を(　2　)演算子という。
c. sizeof演算子の返す型は(　3　)型である。
d. C言語では、「a = a + 2」を「a (　4　) 2」と表現できる。
e. 不等号の演算では、式が偽である場合、式の値は(　5　)となる。
f. インクリメント演算子は(　6　)、デクリメント演算子は(　7　)で表す。
g. *(乗算)演算子は、+(加算)演算子より(　8　)が高い。
h. 同じ優先順位の演算子があった場合、(　9　)に従って評価される。
i. 「op1 ? op2 : op3;」は、op1が真のとき、(　10　)が評価される。

※解答は付録CD-ROMに収録

第４章のまとめ

1 演算子はオペランドを伴って式を構成する。

2 式は評価されてから値を持つ。

3 演算子には優先順位と結合規則が決められている。

4 データ型によって演算結果が変わる。

5 インクリメント演算子（++）は変数を１加算、デクリメント演算子（--）は１減算する。

6 sizeof演算子は式や変数のサイズを返す。

7 不等号演算子は、正しいときは1、正しくないときは0を返す。

8 等価演算子「==」は左右のオペランドが等しいときに１を返す。

9 非等価演算子「!=」は左右のオペランドが等しくないときに１を返す。

10 代入演算子と各種演算子を合わせた演算子がある。

11 キャスト演算子を使うとデータの型を変換できる。

12 ビット演算子は、ビット単位の演算ができる。

13 条件演算子は、条件によって処理を変えることができる。

第5章 制御文

今までのプログラムは、ソースファイルに書かれたコードが上から順に実行されていました。しかし、その時々の状況に応じて実行する順番を変更したり、同じことを何度も繰り返して実行させることができます。このようにプログラムの実行順序を変える文を**制御文**(control statement)といいます。

5.1 if文

「ある条件を満たしたとき(条件が真であるとき)にのみ、実行されるようにしたい！」

はい、簡単です。if文は次のような形になります。

```
if (条件) {
    文1;
    文2;
    ...
}     /* ←セミコロンは不要! */
```

条件が真である場合のみ、文 1，文 2，… とブロック内の文が実行されます。ここでいう「条件」のことを**制御式**（controlling expression）といい、制御式が真のとき実行される文を**真文**（true statement）といいます。

さて、ここで重大な注意があります！！！！ 蛍光ペンを用意してください！！ 真文が実行されるのは制御式の値がいくつのときでしょうか？

「真は 1 だから、1 のときだろう！」

ぶーーーーー（間違いブザーの音です）。もちろん制御式の値が 1 のときは真文が実行されます。しかし、0 以外であれば真文は実行されるのです。C 言語で「真」という場合、これが「1」の場合と「0 以外」を指す場合があります。

「偽」は常に 0 です。どのように使い分けられているのでしょうか。真を 1 とするのは、論理演算（ビット単位ではない）の結果のときです。真を 0 以外とするのは、それ以外の式のときです。

```
if (a = 2) {
    文 1;
}
```

a = 2 は「a に 2 を代入する」という意味で、式の値は 2 です。2 は 0 以外なので文 1 は実行されます。次の例を見てください。

```
a = 2;

if (a == 2) {
    文 1;
}
```

a には 2 が代入されています。(a == 2) は論理演算で、a と 2 が等しいので真となり、式の値は 1 ですね。したがって 0 以外なので文 1 が実行されます。たいていの入門書はこのあたりのことをはっきり書いていません。これで、本書の読者はかなり賢くなったはずです。

ブロック内の文が 1 つのみの場合は

```
if (条件)
    文 1;
```

Column

インデント

{ }で囲まれた部分をブロックといいました。ブロック内の文は行頭より字下げをして（インデントして）書くと見やすくなります。

```
if (a == b) {
printf("...");
printf("....");
}
```

と書くよりも

```
if (a == b) {
    printf("....");
    printf("....");
}
```

と書いたほうが見やすく、間違いも発見しやすくなります。エディタで入力するときはTabキーを押してから入力します。通常タブは８文字に設定されていることが多いのですが、これではソースコードが右に行きすぎてかえって見づらくなります。通常は４文字に設定しておきます。

また、main関数の中身もブロック内に書かれているので、当然次のように字下げをします。

```
int main()
{
    ...;
    ...;
}
```

コンパイラに付属の専用エディタは自動的に字下げをしてくれるので便利です。

というように、{}を省略できます。このように書いた場合、条件を満たしたときに文1だけでなく文2も実行するようにあとでプログラムを変更した場合、つい

```
if (条件)
    文1;
    文2;     /* この文は常に実行される！ */
```

と書いてしまうかもしれません。if (条件)だけで{ }が省略された場合、次の1文のみがifの対象となります。したがって、文2は条件の真偽にかかわらず実行されてしまいます。慣れないうちは、すべてに{ }をつけたほうが無難ですね。

ちょっとif文とは離れますが、今までのプログラムは面白くありませんでしたね。筆者が言っているのですから間違いありません！ なぜでしょう？ それは、プログラム側から一方的に表示をするのみのプログラムだったからです。ユーザーから何か入力できたら、プログラムの幅もぐっと広がるでしょう。ここではscanf関数を紹介します。

scanf関数

標準入力から整形済みデータを読み込みます。

```
int scanf(
    const char *format [,argument]...
);
```

実は、この関数を使うにはまだ重要な知識が足りませんので、次のように機械的に使ってください。

```
数値型 変数;
scanf("変換指定子", &変数);
```

たとえば、

```
int n;
scanf("%d", &n);
```

とすれば、ユーザーがキーボードから整数を入力できます。そして、入力された数値は変数 n に代入されます。変換指定子は入力する内容により次の表 5-1 のようになります。

表 5-1　scanf 関数の変換指定子

変換指定子	入力
%c	文字入力
%d	10 進整数
%f	float 型浮動小数点数
%lf	double 型浮動小数点数
%s	文字列
%p	ポインタ
%o	8 進入力
%x	16 進入力
%u	符号なし 10 進整数

「n の前についている&って何？」

はい、これについては 7.1 節でいやというほど解説しますので、今は「こういうふうに書くんだなーー」程度にしておいてください。

```
/* scan01.c */

#include <stdio.h>

int main()
{
    int seisu;

    printf("整数値を入力してください----");
    scanf("%d", &seisu);
    printf("あなたの入力した数値は %dですね¥n", seisu);

    return 0;
}
```

scanf 関数で入力値を取得

9行目の「printf("....");」のダブルクォーテーションの中身の最後に「¥n」がついていない点に注意してください。改行を防いでいるのですね。実行結果を見てみましょう。

```
整数値を入力してください----125
あなたの入力した数値は 125 ですね
```

どうです。ユーザーの入力が、表示される文章に反映されると感動しますね。これで、やっとプログラムらしいプログラムが書けるようになりました。整数値を入力してください、といっているのに小数や文字列を入力してはいけません。

重要

scanf 関数を使うと、入力値を取得できる。

さて、本題の if 文を使ったプログラムを作ってみましょう。次のようなプログラムを作りたいと思います。これは、年齢により入場料が異なるテーマパークの料金を表示するプログラムです。

1. 年齢を尋ねる
2. 6歳未満なら無料
3. 6歳以上なら 1000 円

まず、年齢を入れておく変数を作っておきましょう。名前は age にでもしておきます。型は int 型でよいでしょう。マイナスの年齢はないので unsigned int 型にしましょうか。あるいは、年齢はせいぜい 100 歳くらいまでなので short 型でも十分かもしれませんね。でも、使い慣れている int 型にしておきましょう。「6歳未満」を表現するには「age < 6」とします。では、プログラムを書いてみましょう。

```c
/* nyujoryo.c */

#include <stdio.h>

int main()
{
    int age;

    printf("このたびは「猫でもランド」に¥n"
           "ご来園ありがとうございます。¥n");
    printf("---------------------------------¥n");
    printf("年齢を入力してください---");
    scanf("%d", &age);

    if (age < 6) {
        printf("入場料は無料です¥n");
    }
    if (age >= 6) {
        printf("入場料1000円を頂きます¥n");
    }
    printf("---------------------------------¥n");
    printf("では、ごゆっくりお楽しみください¥n");

    return 0;
}
```

ageが6未満の場合

ageが6以上の場合

実行すると次のようになります。

```
このたびは「猫でもランド」に
ご来園ありがとうございます。
---------------------------------
年齢を入力してください---18
入場料1000円を頂きます
---------------------------------
では、ごゆっくりお楽しみください
```

ちょっとプログラムらしくなってきましたね。さて、このプログラムについてもう少し考えてみましょう。年齢は、6歳未満以外の人は全員6歳以上ですね。したがって次のように考えることもできます。

6歳未満のとき表示する文章は○○○○○
それ以外のときは□□□□□

これをプログラム的に書くには、else を使います。

```
if (条件 X) {
    文 1;
    文 2;
} else {
    文 3;
    文 4;
}
```

条件Xが真であると文1，文2が実行されます。条件Xが真でないとき、文3，文4が実行されます。このように条件が偽のとき実行される文(文3，文4)を**偽文**(false statement)といいます。ここでも、{ }の中身に文が1つしかない場合は{ }を省略できます。

```
if (条件) {
    文 1;
    文 2;
} else
    文 3;
```

では、先ほどのプログラムを書き換えてみましょう。

```
/* nyujoryo02.c */

#include <stdio.h>

int main()
{
    int age;
```

```
    printf("このたびは「猫でもランド」に¥n"
           "ご来園ありがとうございます。¥n");
    printf("--------------------------------¥n");
    printf("年齢を入力してください---");
    scanf("%d", &age);

    if (age < 6) {
        printf("入場料は無料です¥n");
    } else {
        printf("入場料 1000円を頂きます¥n");
    }
    printf("--------------------------------¥n");
    printf("では、ごゆっくりお楽しみください¥n");

    return 0;
}
```

ageが6未満の場合

それ以外の場合

実行結果は前のプログラムとまったく同じです。

もう少し、条件を複雑にしてみましょう。

1. 6歳未満は入場料無料
2. 6歳未満でなくても「猫でも学園」の生徒なら無料
3. 6歳未満ではなく「猫でも学園」の生徒でもないときは1000円

これには、次のようなelse ifを使うと便利です。

```
if (条件X) {
    文1;
    文2;
} else if (条件Y) {
    文3;
    文4;
} else {
    文5;
    文6;
}
```

条件Xが真であれば文1，文2が実行されます。条件Xが偽で条件Yが真であれば文3，文4が実行されます。さらに、条件Yも偽であれば文5，文6が実行されます。では、プログラムを作ってみましょう。

```c
/* nyujoryo03.c */

#include <stdio.h>

int main()
{
    int age;
    int is_seito;    /* 生徒なら1、そうでないなら0 */

    printf("年齢を入力してください---");
    scanf("%d", &age);
    printf("「猫でも学園」の¥n生徒さんですか(Yes:1, No:0)---");
    scanf("%d", &is_seito);

    if (age < 6) {
        printf("入場料は無料です¥n");
    } else if (is_seito == 1) {
        printf("入場料は無料です¥n");
    } else {
        printf("入場料は1000円です¥n");
    }
    return 0;
}
```

ageは6未満ではないが、is_seitoが1の場合

実行結果を見てみましょう。

```
年齢を入力してください---18
「猫でも学園」の
生徒さんですか(Yes:1, No:0)---1
入場料は無料です
```

重要

条件によって処理を分けるには、if、else if、else を使う。

5.2 関係演算子と論理演算子

if 文の条件式に使われる演算子には、

==, !=, >, <, >=, <=

などがありました。これらを**関係演算子**(relational operator) といいます。「a が6に等しくない」という条件を作るには

```
a != 6
```

とすればよいですね。aが6に等しくないとき、この式の値は1(真)となります。しかし、もっと複雑な条件を作りたいことがよくあります。

例1：「aが6よりも大きい」かつ「aが10以下」
例2：「aが100より大きい」または「aが10より小さい」
例3：「aが200以上ではない」

「かつ」は&&で表します。

A && B

と書けばAもBもともに真であるとき、式の値は真となります。それ以外は偽となります。&&のことを**積結合演算子**(logical AND operator) とか**条件AND演算子**といいます。

「または」は || で表します。

A || B

AかBのどちらか一方（あるいは両方）が真であれば、式の値は真（1）となります。偽（0）となるのはA，Bともに偽のときのみです。||のことを**和結合演算子**（logical OR operator）とか**条件OR演算子**といいます。

さて、「!」演算子というものがあります。これは、

```
!A
```

とした場合、Aが真なら偽、偽なら真になります。これを**否定演算子**（negation operator）」といいます。

また、&&，||，!演算子のことを**論理関係演算子**（logical relational operator）といいます。

さて、論理関係演算子と似たものに**ビット論理演算子**（bitwise logical operator）というものがありましたね（4.10節を参照）。

&&，||，!に対応して&，|，~ があります。論理関係演算子とビット論理演算子をあわせて**論理演算子**（logical operator）といいます。

先のnyujoryo03.cプログラムを次のような考え方で書き直してみましょう――入場料が無料となるのは「年齢が6歳未満」または「猫でも学園の生徒である」ときである。それ以外は1000円。

```
/* nyujoryo04.c */

#include <stdio.h>

int main()
{
    int age;
    int is_seito;     /* 生徒なら1そうでないなら0 */

    printf("年齢を入力してください---");
    scanf("%d", &age);
    printf("「猫でも学園」の¥n生徒さんですか(Yes:1, No:0)---");
    scanf("%d", &is_seito);

    if (age < 6 || is_seito == 1) {
        printf("入場料は無料です¥n");
```

「ageが6未満」または「is_seitoが1」の場合に真

Column　&&は無駄な評価を行わない

「A && B」について考えてみます。AもBも真のとき、この式の値は1となります。もし、Aが偽であればBの真偽に関係なく式の値は0となります。この式ではまずAが評価され、もし0であればBは評価されません。無駄を省いているわけですね。

次のプログラムを実行するとどうなるでしょうか。

```
/* logical01.c */

#include <stdio.h>

int main()
{
    int a = 2, b = 3;

    if (a == 5 && b++)
        printf("真文です¥n");
    printf("b = %d¥n", b);

    return 0;
}
```

「a == 5」は偽ですね。したがってb++は評価されません。したがって実行結果は

```
b = 3
```

となります。しかし、

```
if (a == 2 && b++)
```

と変えてやるとどうでしょう。「a == 2」が真であるためb++が評価され、実行結果は

```
b = 4
```

となります。同じようなことが「A || B」についてもいえます。Aが真ならBを評価する必要はないですね。

```
    } else {
        printf("入場料は 1000円です¥n");
    }
    return 0;
}
```

5.3 switch 文

条件によって実行する処理が異なる場合 if 文を使いました。しかし場合分けをする条件が多数ある場合、if 文では複雑になってしまうことがよくあります。ユーザーに 1 から 10 までの整数値を入力してもらい、それぞれの値によって処理内容を変える場合を想像してみてください。入力された数値が変数 n に収められていたとすると ……

```
if (n == 1) {
    .....;
} else if (n == 2) {
    .....;
} else if (n == 3) {
    .....;
```

というような感じになり、間違いも起こりやすくなります。そこで、switch 文が登場します。

```
switch (制御式) {
    case 事例式:     /* 事例式の後にはコロン */
        文;
        break;
    case 事例式:
        文;
        break;
```

```
    case 事例式:
        文;
    ......
    default:
        文;
        break;
}     /* ←セミコロンは不要! */
```

というような形になります。

```
switch (n) {
    case 5:
        /* nが5のとき処理する内容 */
        ....;
        break;      /* nが5であるときの処理が終わったら
                       switch文から抜ける */
    case 6:
        /* nが6のとき処理する内容 */
        ....;
        break;      /* nが6であるときの処理が終わったら
                       switch文から抜ける */
    default:
        /* nが5, 6以外のとき処理する内容 */
        ....;
        break;      /* 最後のbreak文はなくてもよいが
                       一応つけておく */
}
```

この例ではnの値が5, 6, その他のときのswitch文の形を示しました。

```
case 事例式:
```

を**caseラベル**と呼びます。事例式の後ろにはセミコロンではなくコロンがついていることに注意してください。

```
default:
```

を**defaultラベル**と呼びます。

　また、事例式は整数もしくは文字の定数でなくてはいけません。では、簡単な例を見てみましょう。

```
/* switch01.c */

#include <stdio.h>

int main()
{
    int n;

    printf("1から5までの整数を入力してください。----");
    scanf("%d", &n);

    switch (n) {
        case 1:
            printf("あなたは一番小さい数字を入力しましたね¥n");
            break;
        case 2:
            printf("今日はよいことがあるでしょう¥n");
            break;
        case 3:
            printf("中庸の徳です¥n");
            break;
        case 4:
            printf("4は幸せの4¥n");
            break;
        case 5:
            printf("あなたは5を選びました¥n");
            break;
        default:
            printf("1から5までの整数を入力してください¥n");
            break;
    }

    printf("これでおしまいです¥n");

    return 0;
}
```

入力されたnの値によって分岐

実行結果を見てみましょう。

```
1から5までの整数を入力してください。----3
中庸の徳です
これでおしまいです
```

さて、switch 文の中で break 文を忘れたらどうなるでしょうか。

```
switch (n) {
    case 1:
        文 1;
    case 2:
        文 2;
    default:
        文 3;
}
```

n が 1 であるとすると、まず文 1 が実行されます。しかし、break 文がないので、switch 文を抜けずに、次の文 2 が実行されます。さらに、文 3 も実行されてしまいます。これでは、何が何だかわからないですね。しかし、この性質を逆手にとったプログラムも可能です。n が 1 のときも 2 のときも同じ処理をする場合はどうでしょうか。

```
switch (n) {
    case 1:
    case 2:
        処理 1;
        break;
    case 3:
        処理 3;
        break;
    default:
        処理 4;
        break;
}
```

nが1であっても2であっても処理1が実行されます。ユーザーからアルファベットを1文字入力してもらい、この文字により処理を変える場合を考えます。Aと入力をした場合にある処理をするとします。ユーザーが小文字のaを入力してもAのときと同じ処理をさせたほうが親切ですね。

では、キーボードから1文字入力を受け取るにはどうしたらよいでしょうか。scanf関数が使えそうですね。今までは、数値を受け取って変数nに代入するのに

```
scanf("%d", &n);
```

とやってきました。同様に

```
char ch;
...
scanf("%c", &ch);
```

とやればよいですね。

しかし、ちょっと問題があります。1文字を受け取る関数の場合、内部的にバッファがあってこれに文字をため込んで、Enterキーが押されたときに処理が開始される場合が多いのです。たとえば「S」キーを押して、Enterキーを押したとします。内部のバッファには 'S' 文字と '¥n'（改行）文字が蓄えられています。

そうすると、次に1文字入力の関数が呼ばれたときに、勝手に '¥n' が入力されたと判断されます。そうすると、プログラム上では意図しない動作が起こることになります。ループ内で何度も1文字入力の関数を呼んだときによくこの現象が起こり、プログラマを悩ませます。

キーボードから1文字を読み取るには、scanf関数の他にgetchar関数もよく使われます。

getchar関数

標準入力から文字を読み込みます。

```
int getchar(void);
```

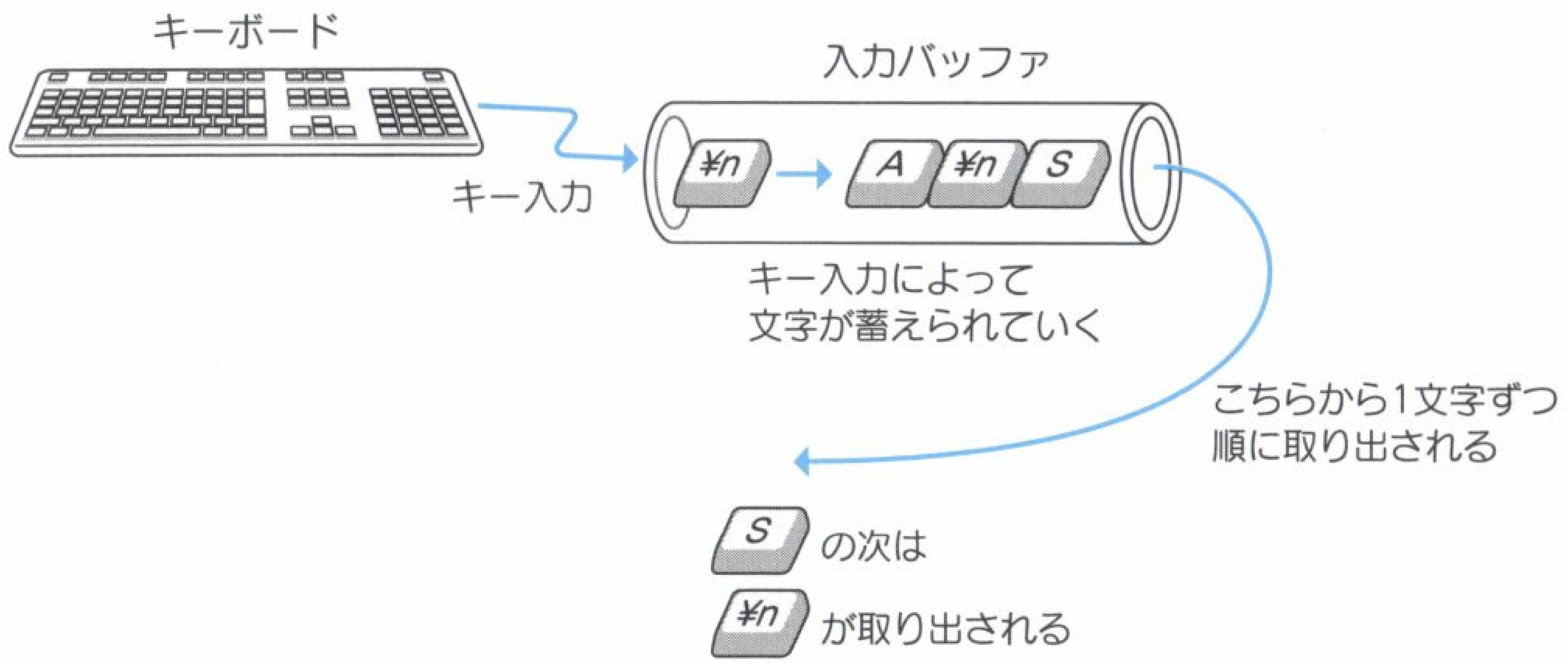


図 5-1　**入力バッファ**

```
c = getchar();
```

とすると、入力した文字がcに代入されます。

次は、メニューを表示して、希望のメニューの文字入力をしてもらうプログラムです。ユーザーが、'X' または 'x' を入力しない限りプログラムは終了しません。

```
/* switch02.c */

#include <stdio.h>

int main()
{
    int c;
    int end = 0;

    while (1) {
        printf("***** MENU ****¥n");
        printf("A:ゲーム¥n");
        printf("B:音楽¥n");
        printf("C:表計算¥n");
        printf("X:終了¥n");
        printf("****************¥n");
        printf("----->");

```

```
        while (1) {
            c = getchar();
            if (c != '¥n')      /* cが'¥n'でないなら */
                break;          /* whileループを抜ける */
        }

        switch (c) {
            case 'A':
            case 'a':
                printf("¥nゲームばかりしないでください¥n¥n");
                break;
            case 'B':
            case 'b':
                printf("¥n音楽が選択されました¥n¥n");
                break;
            case 'C':
            case 'c':
                printf("¥n表計算が選択されました¥n¥n");
                break;
            case 'X':
            case 'x':
                end = 1;
                break;
            default:
                printf("¥n不正な入力です¥n¥n");
                break;
        }
        if (end == 1)     /* endが1なら */
            break;        /* whileループを抜ける */
    }
    printf("¥n終了です¥n");
    return 0;
}
```

実行結果は次のようになります。

```
***** MENU ****
A:ゲーム
B:音楽
C:表計算
X:終了
****************
----->a

ゲームばかりしないでください

***** MENU ****
A:ゲーム
B:音楽
C:表計算
X:終了
****************
----->x

終了です
```

さて、19～23行目の

```
while (1) {
    c = getchar();
    if (c != '¥n')
        break;
}
```

に、注目してください。

ユーザーがAキーを押してEnterキーを押すと、入力バッファには 'A' と '¥n' が蓄えられると書きました。そこで、cが '¥n' でなくなるまでgetchar関数を呼んでいます。cが '¥n' でなくなったらループを抜けて次の処理に移ります。これで、希望通りの動作になったはずです。なお、

```
c = getchar();
```

のcはint型の変数です。不思議に思われるかもしれませんが、文字は数字なのです。8.4節で詳しく解説します。また、whileループに関しては5.5節で解説します。

5.4 for文

for文を使うと希望の回数だけ繰り返して実行ができます。

```
for (初期設定式; 継続条件式; 再設定式) {
    文 1;
    文 2;
    ...
}      /* ←セミコロンは不要! */
```

具体的には次のように使います。

```
for (i = 1; i < 10; i++) {
    文 1;
    文 2;
    ...
}
```

1. 初期設定式は「i = 1」です。これは、最初のみ評価されます。
2. このとき継続条件式(i < 10)は真です。
3. 条件式が真なので文1, 文2が実行されます。
4. 再設定式が評価されます(まだ1増えない)。
5. 次に条件が評価されます(iは1増えて2となる)。
6. 条件式が真なので文1, 文2が実行されます。
7. 再設定式が評価されます(iは2のまま)。
8. 条件式が評価されます(iは3になる)。
9. 条件式は偽なので文1, 文2は実行されません。

なお、{}内に1つの文しかないときは{}を省略できます。

```
/* for01.c */

#include <stdio.h>

int main()
{
    int i;

    for (i = 1; i < 10; i++) {
        printf("%d回目の繰り返しです¥n", i);
    }

    return 0;
}
```

変数iを1ずつ増やしながら
i<10が偽になるまで繰り返す

実行結果はどのようになるでしょうか。

```
1回目の繰り返しです
2回目の繰り返しです
3回目の繰り返しです
・・・
8回目の繰り返しです
9回目の繰り返しです
```

次のプログラムがどのように動作するか予想してみてください。

```
/* for02.c */

#include <stdio.h>

int main()
{
    int i;

    for (i = 0; i <= 20; i += 2)
```

```
            printf("i = %d¥n", i);

        return 0;
}
```

条件式を満たしたときに実行する文が1つだけなので{ }は省略してあります。

```
printf("i = %d¥n", i);
```

が実行された後、再設定式によりiは2ずつ増えますね。したがって実行結果は次のようになります。

```
i = 0
i = 2
i = 4
i = 6
i = 8
i = 10
i = 12
i = 14
i = 16
i = 18
i = 20
```

さて、for文の中にさらにfor文を書くことができます。形の上では、

```
for(....) {
    for(....) {
        ....;
    }
}
```

というようなことになります。大事なことは、どんな順番で繰り返しが起こるかということです。実験してみましょう。

```
/* for03.c */

#include <stdio.h>

int main()
{
    int i, j;

    for (i = 0; i < 4; i++) {
        printf("外側のfor文の1, i = %d¥n", i);

        for (j = 0; j < 3; j++) {
            printf("¥t内側のfor文, j = %d¥n", j);
        }

        printf("外側のfor文の2, i = %d¥n", i);
    }
    return 0;
}
```

外側のループ

内側のループ

実行結果を見てみましょう。

```
外側のfor文の1, i = 0
        内側のfor文, j = 0
        内側のfor文, j = 1
        内側のfor文, j = 2
外側のfor文の2, i = 0
外側のfor文の1, i = 1
        内側のfor文, j = 0
        内側のfor文, j = 1
        内側のfor文, j = 2
外側のfor文の2, i = 1
外側のfor文の1, i = 2
        内側のfor文, j = 0
        内側のfor文, j = 1
        内側のfor文, j = 2
外側のfor文の2, i = 2
```

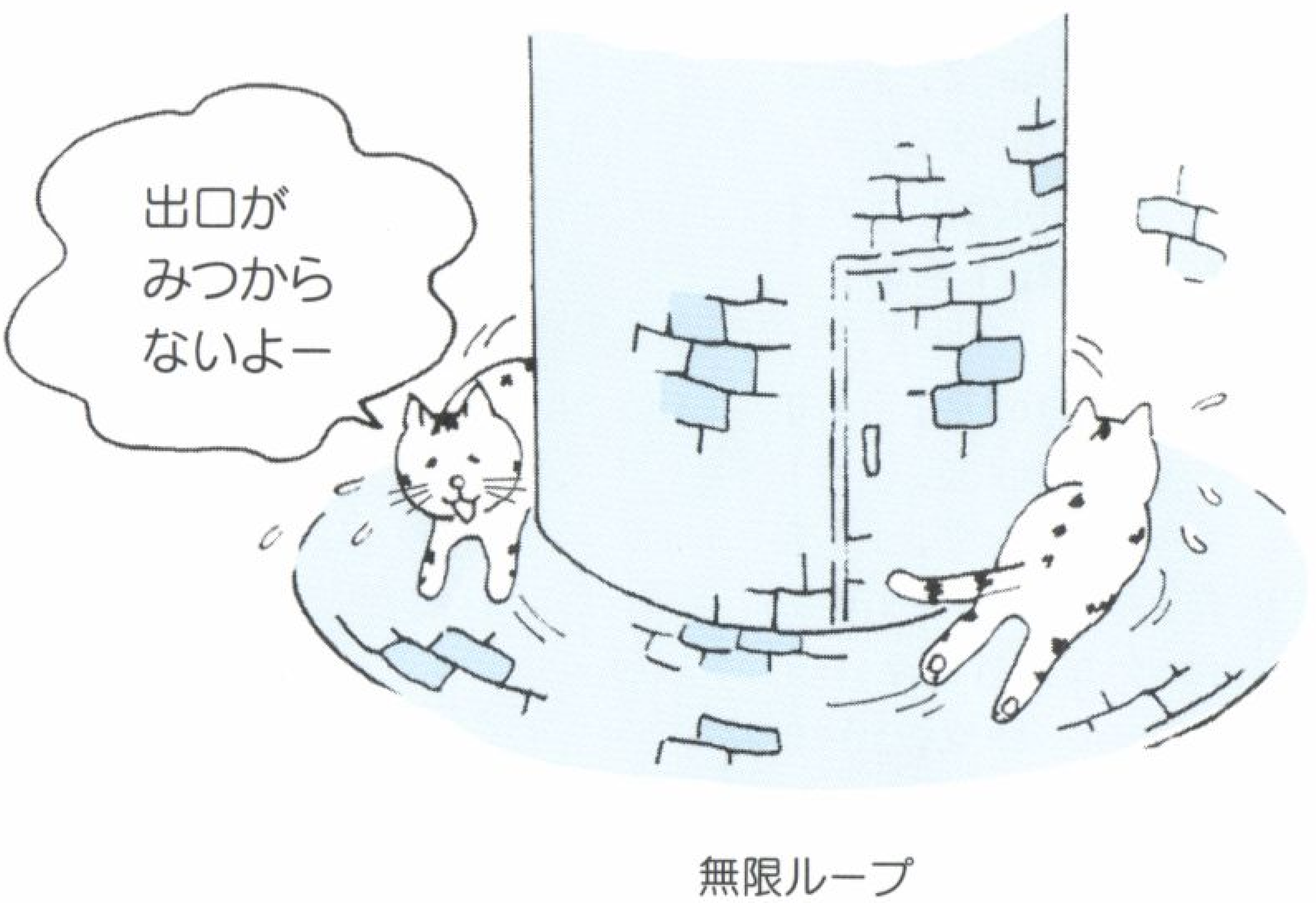
```
外側のfor文の1, i = 3
        内側のfor文, j = 0
        内側のfor文, j = 1
        内側のfor文, j = 2
外側のfor文の2, i = 3
```

外側のfor文で、iは0から3まで1ずつ増加します。それぞについて内側のfor文で、jが0から2まで回ります。意味がわかったでしょうか。

さて、for文を使うプログラムの注意点は何でしょうか。もし、何回繰り返しが起こっても継続条件式が偽にならなかったらどうでしょうか。いつまでたってもプログラムは終了しませんね。こういうのを、「**無限ループ**に陥った」といいます。こんなプログラムを走らせると、システムがハングアップする危険がありますので注意が必要です。

for文で継続条件式を省略するとどうなるでしょうか。これは無限ループになります。無限ループから脱出はできるのでしょうか。

いくつかの方法があります。簡単なのは**break文**を使うことです。すでに使ってきましたが、break文はswitch、while、do～while文を終了させます。



無限ループ

Column　for文を用いたグラフィックス

for文を使って、もう少し視覚に訴えるプログラムを作りましょう。やはり、for文の中にfor文が出てきます。ここではcos関数を使います。cos(ラジアン)とすると、コサインの値が求まります。cos関数を使うにはmath.hをインクルードする必要があります。

```
001: /* cos01.c */
002:
003: #include <stdio.h>
004: #include <math.h>    /* cos()を使うので必要 */
005:
006: int main()
007: {
008:     int i, j, genbun = 5;
009:     double rad, cosvalue, pai = 3.14;
010:
011:     printf("角度¥tcos値¥tイメージ¥n");
012:
013:     for (i = 0; i < 60; i++)
014:         printf("-");
015:
016:     for (i = 90; i >= 0; i -= genbun) {
017:         rad = (pai * i) / 180;
018:         cosvalue = cos(rad);
019:         printf("¥n%2d¥t%f ", i, cosvalue);
020:         for (j = 1; j <= (int)(cosvalue * 40); j++)
021:             printf("*");
022:     }
023:     printf("¥n");
024:     return 0;
025: }
```

for文を3回使っています。最初のfor文(13行目)でprintf関数を使って「-」を60回出力しています。次のfor文(16行目)では、iが90から0になるまで毎回5ずつ減少させています。さらに、このfor文の中にさらにfor文(20行目)がある点に注意してください。

これで、cosvalueに対応した数だけ「*」印が出力されます。もし、cosvalueが1.0なら「*」は40個出力されます。結果はごらんの通り。

```
角度  cos値    イメージ
------------------------------------------------------------
90    0.000796
85    0.087905 ***
80    0.174345 ******
75    0.259460 **********
70    0.342602 *************
65    0.423139 ****************
60    0.500460 ********************
55    0.573975 **********************
50    0.643126 *************************
45    0.707388 ****************************
40    0.766272 ******************************
35    0.819330 ********************************
30    0.866158 **********************************
25    0.906401 ************************************
20    0.939753 *************************************
15    0.965960 **************************************
10    0.984823 ***************************************
 5    0.996199 ***************************************
 0    1.000000 ****************************************
```

「おおーーっ！ コサインカーブだ！」

5.5 while 文

while 文は次のように書きます。

```
while (継続条件式) {
    文 1;
    文 2;
    ....
}     /* ←セミコロンは不要! */
```

継続条件式が真（0 以外）である限り、文 1，文 2，… が繰り返されます。最初から条件式が偽であれば、文 1，文 2，… は一度も実行されません。{ }の中の文が 1 つだけであれば、{ }は省略可能です。では、while 文のサンプルプログラムを見てみましょう。

```c
/* while01.c */

#include <stdio.h>

int main()
{
    int i = 10;

    while (i > 0) {
        printf("%2d 回目の実行です¥n", 11 - i);
        i--;
    }
    return 0;
}
```

i>10 が真である限り繰り返す

実行結果は次のようになります。

```
 1回目の実行です
 2回目の実行です
 3回目の実行です
・・・
 9回目の実行です
10回目の実行です
```

while(1)とすると、いつまでたっても継続条件式が真なので無限ループとなります。脱出するにはbreak文を使うという技はすでに前の節でやりましたね。

次に、テストの平均点を出すプログラムを作ってみましょう。点数は0点から100点までの整数とします。生徒の数は1人以上です(当たり前ですが)。平均点を出すには、点数を合計して人数で割ります。合計点を出すにはちょっとした技が必要です。

1. 合計点をsumとし、最初にsumを0にしておきます。
2. 最初の生徒の点数をpとすると合計は「sum = sum + p」で求められます。
3. 次の生徒の点数をpとするとやはり合計は「sum = sum + p」で求められます。

手始めに合計を求めるプログラムを作りましょう。

```
/* sum01.c */

#include <stdio.h>

int main()
{
    /* sum:合計点, p:点数 */
    int sum = 0, p;

    while (1) {
        printf("点数(999で終了)----");
        scanf("%d", &p);
```

```
        if (p == 999)
            break;
        sum = sum + p;
    }

    printf("合計点は %d 点です¥n", sum);

    return 0;
}
```

終了時には、点数として999を入力します。では実行結果を見てみましょう。

```
点数(999で終了)----85
点数(999で終了)----75
点数(999で終了)----55
点数(999で終了)----90
点数(999で終了)----100
点数(999で終了)----999
合計点は405点です
```

しかしsum01.cはwhile文をわざと無限ループにしており、本来のwhile文の使い方ではないですね。では、sum01.cを書き直してwhile文の本来の使い方のプログラムにしてみましょう。ついでに、平均点も求められるようにします。「平均点＝合計点÷人数」です。人数をカウントするために変数nを用意し、whileループ内でインクリメントしています。

```
/* ave01.c */

#include <stdio.h>

int main()
{
    int sum = 0, p = 0, n = 0;
```

```
    while (p != 999) {
        printf("点数(999で終了)---");
        scanf("%d", &p);
        if (p != 999) {
            sum = sum + p;
            n++;
        }
    }
    printf("%d人の合計点は%d点です¥n", n, sum);
    printf("平均点は%5.1f点です¥n", (double)sum / n);

    return 0;
}
```

pが999でない限りwhile文のところでループします。

最初にpに何か数値を入れておかないと、継続条件式のところでまずいことが起こりますね。それで、pは0で初期化しています。

また、pに999が入ったとき、これを点数に合計したり、人数にカウントしてはまずいですね。そこで、if文でpが999以外のとき、合計を計算したり人数をカウントするようにしています。

さて、実行結果は次のようになります。

```
点数(999で終了)---63
点数(999で終了)---95
点数(999で終了)---78
点数(999で終了)---55
点数(999で終了)---47
点数(999で終了)---999
5人の合計点は338点です
平均点は 67.6点です
```

Column for 文による無限ループ

for 文を使って、意図的に無限ループを作ることができます。しかし、本来の for 文の使い方ではありません。「こんなこともできるのだな――」くらいに考えておいてください。

```
/* for04.c */

#include <stdio.h>

int main()
{
    int i = 1;

    for (;;) {          /* 全部省略して無限ループ */

        printf("%2d回目¥n", i);

        if (i >= 10)    /* i が 10以上なら */
            break;      /* for文を抜ける */
        i++;

    }

    return 0;
}
```

実行結果を見てみましょう。

```
 1回目
 2回目
 3回目
...
 9回目
10回目
```

for 文とは関係ないですが「回目」の文字が縦に揃っている点にも注目してください。

5.6

do〜while文

さて、while文では継続条件式によっては、一度も実行されない処理が存在する可能性がありました(前置判定)。

```
while (0) {
    文1;
}
```

こういうプログラムでは、文1は一度も実行されません。しかし、do〜while文(後置判定)ではこのようなことは起こりません。

```
do {
    文1;
    文2;
    ...
} while (継続条件式)
```

のような形になります。まず、文1, 文2, … が実行されます。次に継続条件式が評価され、真ならもう一度、文1から実行されます。継続条件式が偽になるまで繰り返されます。例題を見てみましょう。

```
/* do01.c */

#include <stdio.h>

int main()
{
    int n;

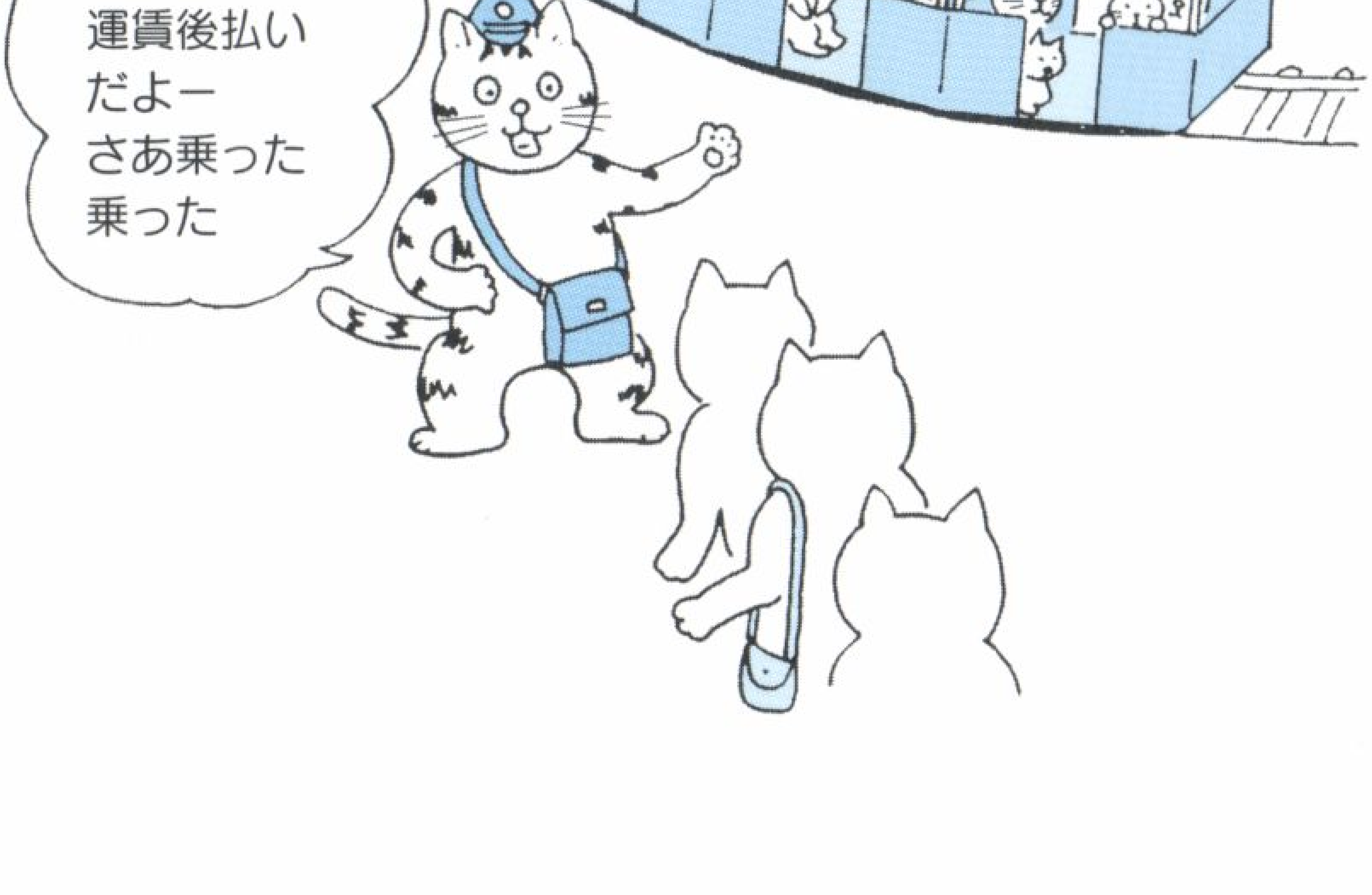
    do {
        printf("整数を入力してください(0で終了)--- ");
        scanf("%d", &n);
        printf("あなたの入力した数字は%dですね¥n", n);
    } while (n);
```

処理の後で繰り返し条件を判定する

```
    return 0;
}
```

nが0になると終了です。実行結果は次のようになります。

```
整数を入力してください(0で終了)--- 10
あなたの入力した数字は10ですね
整数を入力してください(0で終了)--- -3
あなたの入力した数字は-3ですね
整数を入力してください(0で終了)--- 0
あなたの入力した数字は0ですね
```

5.7 break 文と continue 文

break 文はすでに何回も出てきました。これは、switch, for, while, do ～ while 文の実行を終了させるものでした。これとは逆に、ループ内の残りの処理を飛ばしてブロックの先頭に戻って処理を続けるものに **continue 文**があります。では、例題を見てみましょう。

```
001: /* continue01.c */
002:
003: #include <stdio.h>
004:
005: int main()
006: {
007:     int i, j;
008:
009:     for (i = 1; i < 10; i++) {
010:         for (j = 1; j < 10; j++) {
011:             if (i > j)
012:                 continue;
013:             printf("%d X %d = %2d¥n", i, j, i * j);
014:         }
015:         printf("----------------¥n");
016:
017:     }
018:
019:     return 0;
020: }
```

i>jになったらループの先頭(10行目)に

これは、九九の表を作るプログラムです。3×5は5×3と同じなので後者を省略した表です。(i › j)の条件が真であればcontinue 文が実行され、13行目のprintf 関数が省略されて(内側の)ループの先頭(10行目)に戻ります。

実行結果は次のようになります。

```
1 X 1 =  1
1 X 2 =  2
1 X 3 =  3
1 X 4 =  4
1 X 5 =  5
1 X 6 =  6
1 X 7 =  7
1 X 8 =  8
1 X 9 =  9
-----------------
2 X 2 =  4
2 X 3 =  6
2 X 4 =  8
2 X 5 = 10
2 X 6 = 12
2 X 7 = 14
2 X 8 = 16
2 X 9 = 18
-----------------
3 X 3 =  9
3 X 4 = 12
3 X 5 = 15
3 X 6 = 18
3 X 7 = 21
3 X 8 = 24
3 X 9 = 27
-----------------
4 X 4 = 16
4 X 5 = 20
4 X 6 = 24
4 X 7 = 28
4 X 8 = 32
4 X 9 = 36
-----------------
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
-----------------
6 X 6 = 36
6 X 7 = 42
6 X 8 = 48
6 X 9 = 54
-----------------
7 X 7 = 49
7 X 8 = 56
7 X 9 = 63
-----------------
8 X 8 = 64
8 X 9 = 72
-----------------
9 X 9 = 81
-----------------
```

重要

- break文はループを抜ける。
- continue文は、残りの処理を飛ばして、繰り返し処理を継続する。

5.8 goto 文

もうひとつ、ループを抜けるための文としてgoto文があります。これはあまり使われません。

```
goto ラベル名;
```

とすると、ラベル名のところに飛んでいきます。ラベルはgoto文と同じ関数内にただ1つ存在しなくてはいけません。

ラベル名は

```
ABC:
```

のように最後にコロンをつけます。

では、5.5節で作ったwhile01.c（117ページ）をgoto文を使ったプログラムに書き換えてみましょう。

```
001: /* goto01.c */
002:
003: #include <stdio.h>
004:
005: int main()
006: {
007:     int i = 10;
008:
009:     START:
010:     if ( i > 0) {
011:         printf("%2d回目の実行です¥n", 11 - i);
012:         i--;
013:         goto START;
014:     }
015:     return 0;
016: }
```

START ラベル（9行目）に飛ぶ

実行結果は次のようになります。

```
 1回目の実行です
 2回目の実行です
・・・
 9回目の実行です
10回目の実行です
```

goto文は、C言語ではたいへんな嫌われ者です。これを多用すると非常にわかりにくいプログラムとなります。「できれば使わないように」と書いてある入門書も多いようです。

ただ、goto文で書くと非常にすっきりする場合もあります。二重ループから一気に外に出たいときなど、goto文は重宝します。

練習問題

1. 「1＋2＋3＋…＋100」の値を求めるプログラムを、for文を利用して作ってください。
2. 1.と同じプログラムを、while文を利用して作ってください。
3. あるクラスのテストの最低点、最高点、受験者数、平均点を求めるプログラムを作ってください。点数は0点から100点の間の整数値です。平均点は小数第1位まで求めてください。なお、点数の入力が終わったら-1を入力するものとします。また、受験者数が0人のときは、ただちにプログラムを終了するようにしてください。
4. 体重(kg)を身長(m)の2乗で割ったものをBody Mass Index(BMI)といいます。身長と体重を入力してBMIを求めるプログラムを作ってください。また、BMIが22であるような体重が理想的であると考えられています。入力した身長に対して、理想的な体重も表示するようにしてください。

※解答は付録CD-ROMに収録

第5章のまとめ

1. 条件に応じた処理を実行するには、if・else if・elseを使う。
2. 入力値を取得するには、scanf 関数を使う。
3. 関係演算子を組み合わせると、さまざまな条件を作成できる。
4. 条件式の値に応じた処理を実行するには、switch 文を使う。
5. 希望の回数だけ繰り返し処理を行うには、for 文を使う。
6. 継続条件を先に判断してから繰り返し処理を行うには、while 文を使う（前置判定）。
7. 必ず一度は処理を行う場合は、do ～ while 文を使う（後置判定）。
8. 繰り返し処理を終了するにはbreak 文を使う。
9. 残りの処理を飛ばして繰り返し処理を継続するにはcontinue 文を使う。
10. 特定の箇所に飛ぶにはgoto 文を使う。

第6章

関数

すでに**関数**(function)は、何回も使っています。main関数、printf関数などなど。プログラムはすべて、main関数の中に書いていました。この章では、自分で関数を作る方法とその活用法を解説します。

6.1 関数の基本

関数は、よく自動販売機にたとえられます。自動販売機は、お金を入れると、商品が出てきます。中がどうなっているかはわからなくても、お金という入力をすれば、商品というものに姿を変えて出力されますね。

関数を利用するには、関数の名前(自動販売機の名前)と、関数に渡す値(お金)が必要です。

自分で作った関数funcを例にしてみましょう。この関数は、int型の整数を2つ渡すと、その積を教えてくれる(返す)関数だとします。この関数を呼び出すには、次のように書きます。

```
int a, b, c;

a = 10;
b = 20;

c = func(a, b);
```

aやbを**引数**（ひきすう）といいます。関数が返してくれる値を**戻り値**とか**返却値**といいます。

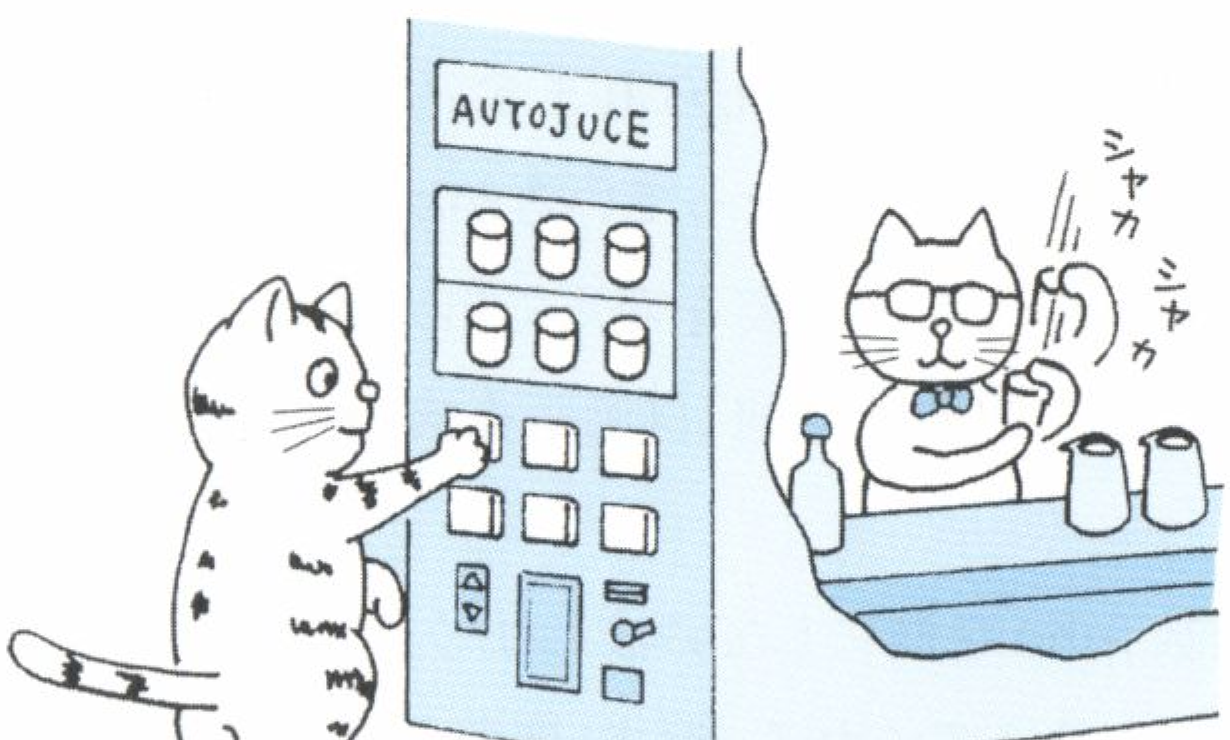
では、実際のプログラムで次のように書くとどうでしょうか。

```
#include <stdio.h>

int main()
{
    int a, b, c;

    a = 10;
    b = 20;
    c = func(a, b);
    printf("%d x %d = %d¥n", a, b, c);

    return 0;
}
```



販売機（関数）の中で何が起こっているか知らなくても
ジュース（結果）は得られる

これは、当然エラーになってコンパイルできません。この時点では、func関数がどのような関数かまったくわからないからですね。そこで、func関数がどのような関数なのか、コンパイラにわかるように定義しなくてはいけません。関数は次のように定義します。

```
戻り値の型 関数名(引数の並び)
{
    やりたいことを書く;

    return 戻り値;
}
```

おや、これはいつものmain関数と同じ形ですね。そうです。main関数も関数なので、当然同じ形をしています。では、自作関数funcを定義してみましょう。

```
int func(int a, int b)
{
    int c;
    c = a * b;

    return c;
}
```

aとbには、呼び出し側からもらってきた値が入っています（図6-1）。

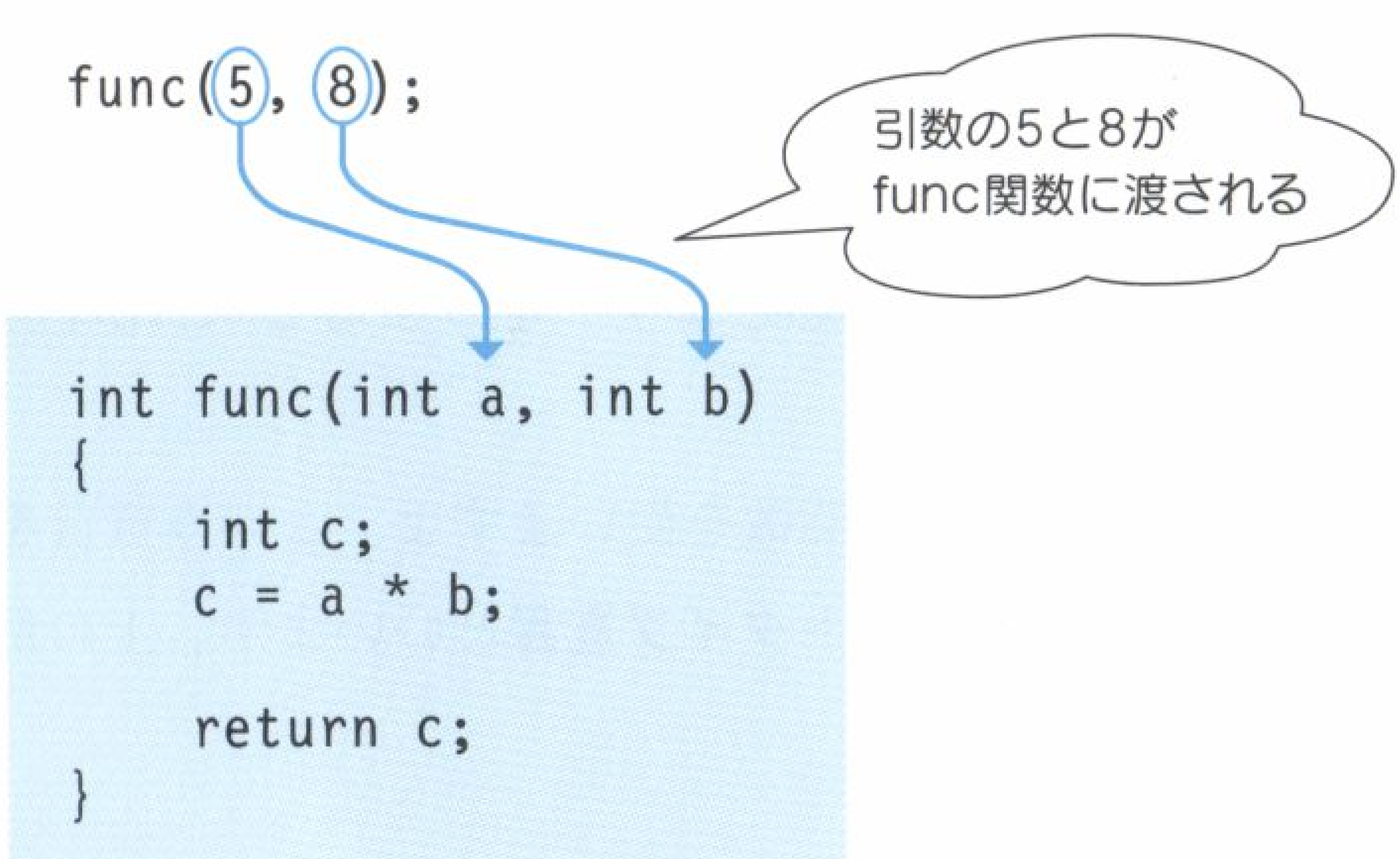


図6-1 引数の受け渡し

この関数の役目は、引数としてもらってきた２つのint型の値から、その積を求めて返してやることでした。そこで、aとbの積をcに代入して、これをreturnしています。

では、これをプログラムに書いてみます。

```
#include <stdio.h>

int main()
{
    int a, b, c;
    a = 10;
    b = 10;
    c = func(a, b);
    prinf("%d x %d = %d¥n", a, b, c);

    return 0;
}

int func(int a, int b)
{
    int c;
    c = a * b;
    return c;
}
```

これは、どうでしょうか。おそらく、コンパイラから警告が出されたか、エラーになったのではないでしょうか。コンパイラは、ソースファイルの先頭から順番に翻訳していきます。そうすると、main関数を翻訳中にfunc関数という、聞いたこともない関数に出くわします。そうすると「こんな関数知らないよ」という苦情を言うわけです。

これを防ぐには、最初にこんな関数を使いますよ、という宣言をしておく必要があります。これを関数の**プロトタイプ宣言**といいます。上の例では

```
int func(int, int);
```

という感じで宣言します。

```
int func(int a, int b);
```

のように宣言してもよいのですが、引数の型がわかれば十分なのでaとかbのような変数名は省略できます。

「あれ、そういえばprintf関数なんかはプロトタイプ宣言していないけど、どうなってるの？」

実は、printf関数のような標準関数でもちゃんとプロトタイプ宣言しているのです。おまじないのように書いていた

```
#include <stdio.h>
```

に秘密があります。printf関数のプロトタイプはstdio.hに記されています。

```
#include ヘッダファイル名
```

とすると、ヘッダファイルという種類のテキストファイルに書いてあることが、そっくりそのままソースファイルにコピーされるのです。includeとは「取り込む」という意味です。まさに、ヘッダファイルがソースファイルに取り込まれているのですね。ヘッダファイルの拡張子は「.h」となります。

また、システムに用意されてあるヘッダファイルは < > で囲みます。一方、自分で作ったヘッダファイルは " " で囲みます。

では、実際に自作関数が正しくコンパイルできて、思ったように動作するのか試してみましょう。

```
/* function01.c */

#include <stdio.h>

/* 自作関数のプロトタイプ宣言 */
int func(int, int);

int main()
{
    int a, b, c;

    a = 10;
```

関数のプロトタイプ宣言

```
    b = 20;
    c = func(a, b);

    printf("%d x %d = %d¥n", a, b, c);

    return 0;
}

int func(int a, int b)
{
    int c;
    c = a * b;
    return c;
}
```

関数呼び出し

関数の実体

実行結果は次のようになります。

```
10 x 20 = 200
```

6.2 引数と戻り値

どうやら、自作関数は正しく動いているようですね。さて、このプログラムで勘違いしやすい点があります。

```
int func(int a, int b)
{
    ...
}
```

とありますが、main 関数で「func(a, b);」と呼び出しているがゆえに func 関

数の引数を「int a, int b」としている、というわけではありません。main 関数からは、単に int 型の値を２つもらってきているという意味ですから、引数名は「a、b」ではなく、任意の名前を使うこともできます。

```
int func(int x, int y)
{
    int kotae;
    kotae = x * y;
    return kotae;
}
```

このとき x や y を**仮引数**（parameter）といいます。一方呼び出し側の func(a, b); の a や b を**実引数**（argument）といいます。

さて、関数はいつも戻り値を返すとは限りません。何も返さない関数もあります。この場合は **void** というキーワードを使います。

```
void 関数名(引数の並び)
{
    ....
    return;    /* returnのみで何も返さない */
}
```

また、引数を持たない関数もあります。

```
戻り値の型 関数名(void)
{
    ...
    return 戻り値;
}
```

のように書きます。引数の場合は、void を省略してもかまいません。

さて、「return 戻り値;」とか「return;」を **return 文**といいますが、この働きは何でしょうか。これには、関数をそこで中断して、呼び出し元に制御を戻す働きがあります。１つの関数の中に複数の return 文があってもかまいません。場合分けをして、そのときの状況に応じて return するようなプログラムも考えられますね。

Column　プロトタイプ宣言を書かずに済ます

さて、関数のプロトタイプ宣言が面倒くさい、という方には次のような方法もあります。

「自作関数の定義を先に書いて、最後に main 関数を書く」

これを、**ボトムアップ方式**といいます。function01.c をボトムアップ式に書くと次のようになります。

```
#include <stdio.h>

int func(int a, int b)
{
    int c;
    c = a * b;
    return c;
}

int main()
{
    int a, b, c;

    a = 10;
    b = 20;
    c = func(a, b);

    printf("%d x %d = %d¥n", a, b, c);

    return 0;
}
```

func 関数を先に書いているのでプロトタイプ宣言を書かなくてよくなります。コンパイラは、ソースファイルを先頭から順番に翻訳していく、ということを考えるとプロトタイプ宣言をしなくてよい理由がわかりますね。

では、main関数で最後にいつも「retun 0;」としていますが、これは誰に値を返しているのでしょうか。この値はシステムに返していると考えてください。このときの戻り値は0でなくてもかまいません。

6.3 変数の寿命とスコープ

次のプログラムを見てください。

```
/* span01.c */

#include <stdio.h>

int add(int);

int main()
{
    int sum;

    sum = add(10);
    printf("sum = %d¥n", sum);

    sum = add(20);
    printf("sum = %d¥n", sum);

    sum = add(100);
    printf("sum = %d¥n", sum);

    return 0;
}

int add(int x)
{
    int gokei = 0;
    gokei += x;
```

```
    return gokei;
}
```

add関数は、呼び出し元からもらった値をgokeiにどんどん加えていくつもりで作った関数です。たとえば

```
sum = add(10);
```

とすればsumの値は10になり、次に

```
sum = add(20);
```

とすれば、sumの値は30になり、次に

```
sum = add(100);
```

とすれば、sumの値は130になることを期待して作ったわけです。では、本当にそうなるのか試してみましょう。

```
sum = 10
sum = 20
sum = 100
```

ありゃ、全然だめですね。これは、どうしたことでしょうか。実は、関数内で宣言した変数には寿命があるのです。関数が制御を戻すたびに(returnするたびに)、変数の値は無効になってしまうのです。

ですから、add関数内のgokeiには、どんどんxの値が合計されているわけではないのです。次にadd関数が呼び出されたときgokeiは、「int gokei = 0;」で初期化されて0になっています。

関数内で変数を宣言すると、その変数のためのメモリが確保されます。returnすると、確保されたメモリは破棄されてしまいます。確保されてから破棄されるまでを、その変数の**寿命**といいます。また、add関数内のgokei変数は、main関数から見えません(読み出したり、書き換えたりできない)。こ

れを変数の**スコープ**(scope，適用範囲)といいます(図6-2参照)。

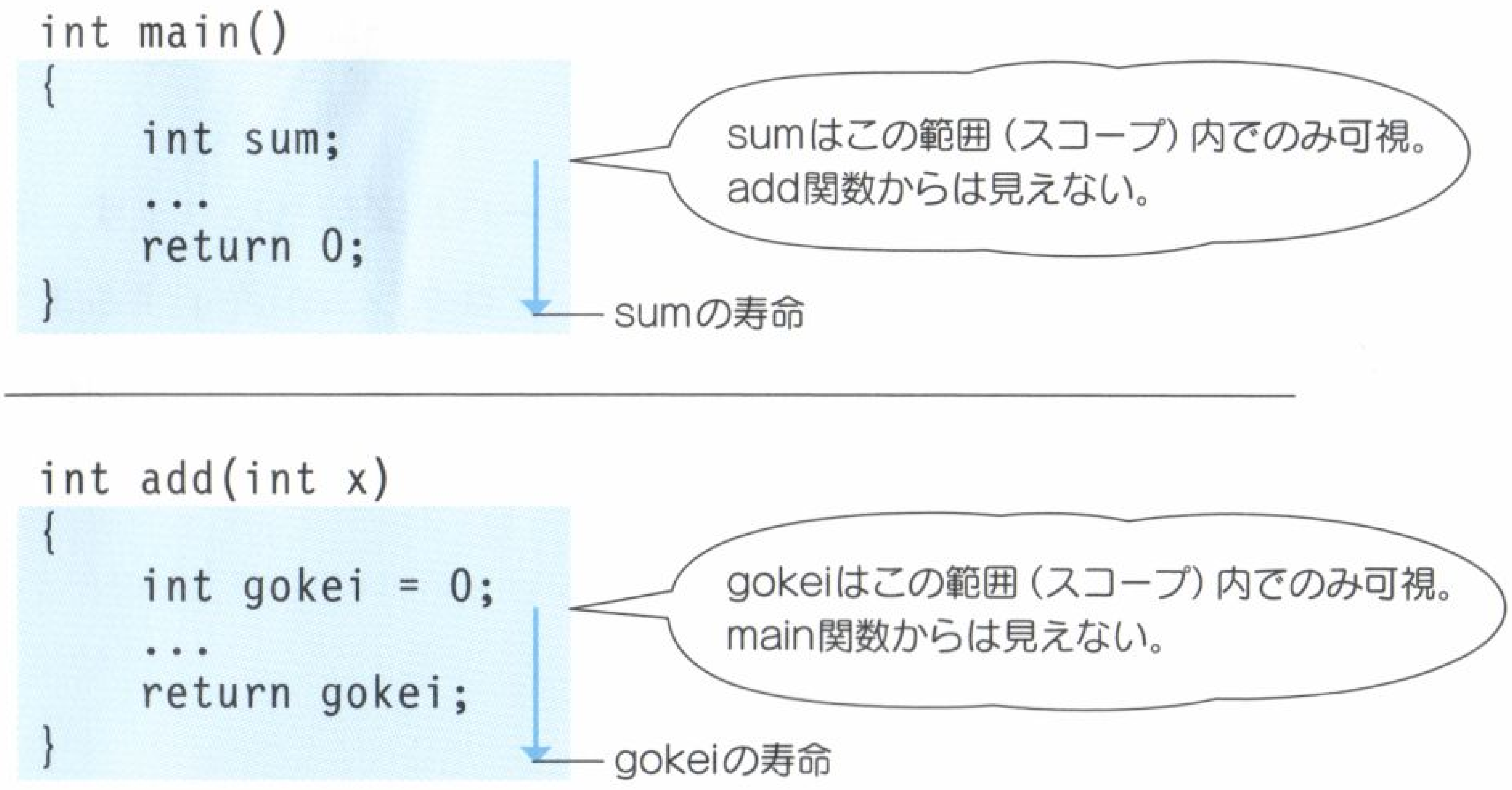


図6-2　変数の寿命とスコープ

では、関数がreturnしても値を保持し続けるようにするには、どうしたらよいのでしょうか。これには、変数に**static**というキーワードをつけます。これは、**記憶クラス指定子**(storage class specifier)と呼ばれるものです。これには、extern，static，auto，registerなどがあります。

autoはその変数が自動変数であることを示します。**自動変数**は関数内もしくはブロック内で宣言された変数で、その関数やブロック内のみで有効です。制御がその関数やブロックを離れたら破棄されます。自動変数はスタック領域と呼ばれるメモリ上に記憶されます。

```
int func()
{
    auto int x;
    {
        auto int y;
        ...
    }
    .....
    return 戻り値;
}
```

変数 x はこの関数が return するまで有効です。変数 y は制御がブロックを離れると無効となります。いずれの変数についても、auto は省略できます。

自動変数は関数内とかブロック内のみで有効なので、**ローカル変数**とも呼ばれます。

これに対して、関数外で変数を宣言すると**グローバル変数**となります。これは、ヒープ領域と呼ばれるメモリ上に記憶され、プログラムが終了するまで値を保持します。また、変数を初期化しなければ自動的に 0 で初期化されます。宣言した場所以降であれば、どこからでも参照したり書き換えたりできます。

```
#include <...>
int func();

int x;

int main()
{
    ...
}

int func()
{
    ...
}
```

この場合、x はグローバル変数で、main 関数、func 関数の両方から利用できます。次の場合はどうでしょう。

```
#include <...>
int func();

int main()
{
    ...
}

int x;
```

```
int func()
{
    ...
}
```

変数xを宣言している位置に注目してください。グローバル変数をこのように宣言すると、func関数内からは見えるが、main関数からは見えません。

また、グローバル変数と同じ名前のローカル変数は使用可能です。この場合、ローカル変数が優先されます。

```
/* local01.c */

#include <stdio.h>

int function();

int i = 10;

int main()
{
    int i = 5;
    {
        int i = 3;
        printf("main関数の中のブロック内:i = %d¥n", i);
    }
    printf("main関数内:i = %d¥n", i);
    function();

    return 0;

}

int function()
{
    printf("function関数:i = %d¥n", i);
    return 0;
}
```

実行結果は次のようになります。

```
main関数の中のブロック内:i = 3
main関数内:i = 5
function関数:i = 10
```

staic変数は、すでに述べたようにプログラムが終了するまで値を保持します。また、宣言時に初期化しないと、自動的に0で初期化されます。では、span01.cを書き換えて、希望通りの動作をするようにしてみましょう。

```c
/* span02.c */

#include <stdio.h>

int add(int);

int main()
{
    int sum;

    sum = add(10);
    printf("sum = %d¥n", sum);

    sum = add(20);
    printf("sum = %d¥n", sum);

    sum = add(100);
    printf("sum = %d¥n", sum);

    return 0;
}

int add(int x)
{
    static int gokei = 0;
    gokei += x;
```

```
    return gokei;
}
```

これは、add関数内のgokei変数をstatic変数にしただけです。では、実行してみましょう。

```
sum = 10
sum = 30
sum = 130
```


どうです。希望通りの結果になりましたね。

extern変数は、関数の外側で宣言し、他のソースファイルからも参照できます。

source01.c

```
#include <...>

int x;

int main()
{
    ...
}
```

source02.c

```
extern int x;

int func()
{
    ...
}
```

このように、ソースファイルを複数に分割してコンパイルすることもできます。

source01.cの変数xはsource02.cでも参照できます。分割コンパイルについては11.6節で解説します。

重要

- 変数には寿命とスコープがある。
- 関数の外で宣言した変数はグローバル変数となる。

6.4 関数の再帰呼び出し

C言語では、関数は自分自身を呼び出すことができます。これを**再帰呼び出し**(recursive call)といいます。再帰呼び出しはスタック領域のメモリを大量に消費しますので、多用は避けたほうがよいでしょう。

main関数からmain関数を呼び出すことも可能です。

```
/* recursive01.c */

#include <stdio.h>

int main()
{
    static int i = 1;

    if (i <= 10) {
        printf("i = %d¥n", i);
        i++;
        main();
    }
    return 0;
}
```

自分自身(main())を呼び出す

実行結果は次のようになります。

```
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
```


再帰呼び出しで例としてよく使われるのが、階乗を求める関数です。n 階乗とは

```
n * (n - 1) * (n - 2) * ... *3 * 2 * 1
```

のことであり、n!と書きます。たとえば4! は

```
4 * 3 * 2 * 1
```

で 24 です。また、0! は 1 と定義されています。

では、階乗を求めるプログラムを作ってみましょう。

```
/* recursive02.c */

#include <stdio.h>

int kaijo(int);

int main()
{
    int i;
    for (i = 0; i < 11; i++)
        printf("%d! = %d¥n", i, kaijo(i));

    return 0;
}
```

```
int kaijo(int n)
{
    if (n == 0)
        return 1;
    else
        return n * kaijo(n - 1);
}
```

このプログラムの意味がわかるでしょうか。

```
kaijo(0);
```

と呼び出された場合「return 1;」が実行されkaijo関数は1を返します。これは、文句はないですね。次に、

```
kaijo(4);
```

という具合に呼び出されるとどうなるでしょうか。

1. まずは「4 * kaijo(3)」がreturnされる。
2. ここでまた、kaijo関数が呼ばれる。
3. そうすると「3 * kaijo(2)」がreturnされる。
4. ここでまた、kaijo関数が呼ばれる。
5. そうすると「2 * kaijo(1)」がreturnされる。
6. またまたkaijo関数が呼ばれる。
7. このとき、kaijo関数は「1 * kaijo(0)」をreturnする。
8. ここで呼ばれたkaijo関数は1を返す。

kaijo(4)の戻り値を見ていくと次のようになります。

```
4 * kaijo(3)
4 * (3 * kaijo(2))
4 * (3 * (2 * kaijo(1)))
4 * (3 * (2 * (1 * kaijo(0))))
4 * (3 * (2 * (1 * 1)))
```

結局kaijo(4)の戻り値は「4 * 3 * 2 * 1 * 1」となるわけです。

では、recursive02.cの実行結果を見てみましょう。

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
8! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

さて、kaijo関数は条件演算子を使って次のように書き直すことができます。

```
int kaijo(int n)
{
    return (n == 0) ? 1: n * kaijo(n - 1);
}
```

練習問題

1. 円の半径を入力すると、戻り値として円の面積を返す関数を作って、実際のプログラムで動作を確かめてください。円周率は 3.14 を使ってください。

2. 「1!, 2!, 3!, 4!, …, 10!」の値を表示するプログラムを作ってください。

3. $F_1 = 1$, $F_2 = 1$, n が 3 以上のとき $F_n = F_{n-1} + F_{n-2}$ で与えられる数列をフィボナッチ数列といい、各項をフィボナッチ数といいます。添え字を引数として与えたときにフィボナッチ数を返す関数を作ってください。また、実際にこの関数を使って n が 1 から 30 までのフィボナッチ数を表示するプログラムを作ってください。

※解答は付録 CD-ROM に収録

第6章のまとめ

1 一定の処理をひとまとまりにして、関数として定義できる。

2 関数に渡す値を引数という。

3 関数から返ってくる値を戻り値という。

4 関数プロトタイプを宣言して、コンパイラに関数の仕様を知らせる必要がある。

5 変数には寿命とスコープ(適用範囲)がある。

6 関数は自分自身を再帰的に呼び出すことができる。

第7章 ポインタ

C言語において「ポインタ」は非常に重要な概念です。ポインタ（pointer）とは、pointするものです。pointは「指し示す」という意味ですね。

つまり、ポインタとは何かを指し示しているものなのです。これじゃあ、なんのことだかわかりませんね。しかしこの章を読み終えたころには、しっかり理解できるはずです。ポインタでつまずいてC言語を投げ出してしまう人も多いようです。ポインタでつまずきそうになったら、またこの章を読み直してください。

ポインタを理解するには、メモリのアドレスについての理解が必要となります。この章ではメモリのアドレスとポインタの関係について解説します。ポインタは非常に強力で有用な道具となりますが、使い方を誤るとプログラムが暴走したり、予想外の結果を生むことになるので注意が必要です。ポインタを利用することでどんなメリットがあるのでしょうか。順を追って解説していきましょう。

7.1 メモリのアドレス

ポインタの解説に入る前に、まずは、ポインタと密接な関係にあるメモリについて解説します。コンピュータがプログラムを実行する際には、必ずプログラムや変数などのデータをメモリ上に置いて処理を行います。

メモリは1バイトずつ区分けされており、これらには0番地、1番地、2番地、…… というような独自の住所が割り当てられています。この住所のことを**アドレス**(address)といいます(図7-1)。

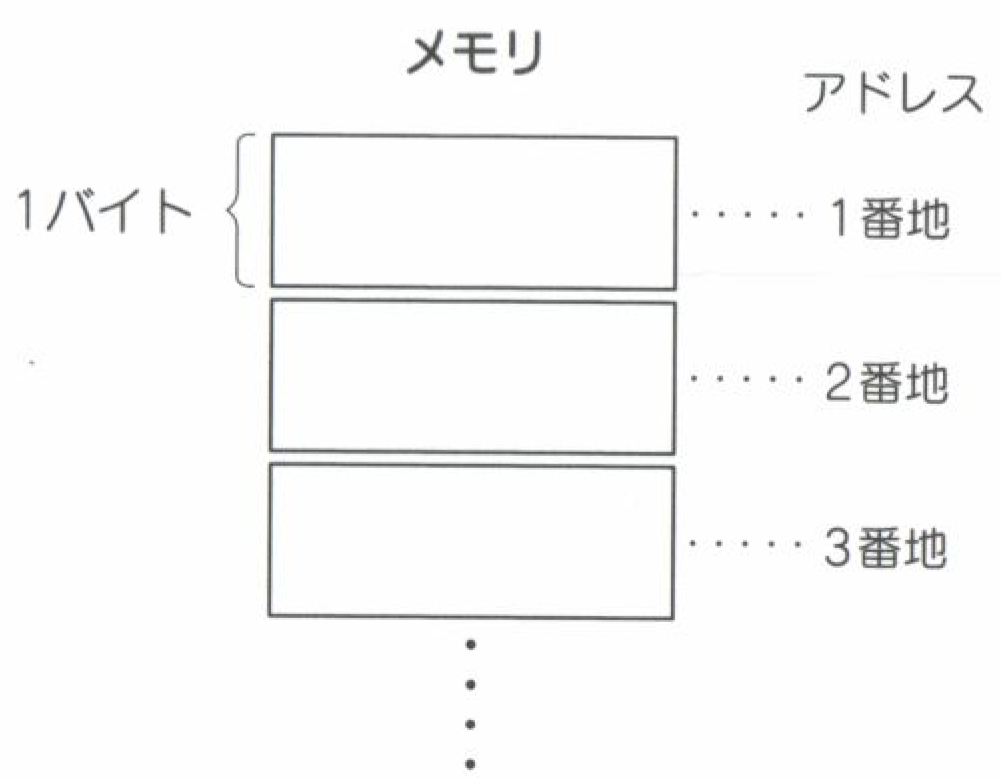


図7-1 **メモリのアドレス**

前章までに何度も変数の宣言・定義や代入などを行ってきましたが、これらはすべてメモリ上のどこかのアドレスに記録されているわけです。変数がどのアドレスにあるか、プログラムを作って調べてみましょう。これはとても簡単で、変数に「&(アンパサンド)」をつけるだけです。

&変数名

たとえば「&a」は変数aが保存されているメモリの番地、つまりアドレスを意味しています。「&」のことを**アドレス演算子**(address operator)といいます。「アンパサンドのアはアドレスのア」と覚えておくと忘れません。

ビット演算のときにも「&」を使いましたが、それとは別です。ビット演算のときは「A & B」と書くとAとBのビット積が求まりますが、この場合はオペラ

ンドがAとBの２つあります。アドレス演算子と間違えることはありません。

それでは実際にいろいろな変数を宣言して、その変数が置かれているメモリのアドレスを、printf関数を使って表示してみましょう。printf関数でアドレスを表す書式指定フィールドは「%p」です。

```
/* address01.c */

#include <stdio.h>

int main()
{
    char c;
    int i;
    double d, e;

    printf("変数 cのアドレスは%pです¥n", &c);
    printf("変数 iのアドレスは%pです¥n", &i);
    printf("変数 dのアドレスは%pです¥n", &d);
    printf("変数 eのアドレスは%pです¥n", &e);

    return 0;
}
```

実行結果は次のようになりました。

```
変数 cのアドレスは0012FED3です
変数 iのアドレスは0012FED4です
変数 dのアドレスは0012FED8です
変数 eのアドレスは0012FEE0です
```

「0012FED3」のように16進数で表示されているのが、変数が置かれているアドレスです。ただし、この結果は環境によって異なります。みなさんが実行したときには、これとは違う結果になるでしょう。

「こんな複雑な数字を見るとめまいがするぞ！」

ご安心ください。実際にプログラムを書くときは、このようなアドレス値をそのままの形で扱うことはまずありません。

本書で使用しているVisual C++ 2008では、char型は1バイト、int型は4バイト、double型は8バイトの大きさです。おそらくほとんどのコンパイラはそのような仕様になっていると思われます。これを踏まえて、実行結果をよく観察してみましょう。

- 変数ｉと変数ｃのアドレスの差は１バイト
- 変数ｄと変数ｉのアドレスの差は４バイト
- 変数ｅと変数ｄのアドレスの差は８バイト

つまり、この場合

- 変数ｃは0012FED3番地の１バイト
- 変数ｉは0012FED4番地から0012FED7番地までの４バイト
- 変数ｄは0012FED8番地から0012FEDF番地までの８バイト

を使っているのです。図7-2を見てください。変数のサイズとメモリアドレスの関係がわかると思います。

7.2 ポインタの基本

変数のアドレスを求めることができるようになりました。このアドレスを変数に代入することはできないのでしょうか。アドレスも単なる整数値なので、変数に代入できそうですね。実は、アドレスを格納するための変数が**ポインタ**なのです。

ポインタを使うためには、一般の変数と同じようにブロックの最初に宣言する必要があります。int型の変数のアドレスを格納するポインタをpiとすると

```
int *pi;
```

というように宣言します。この場合、piは「int型へのポインタ」となります。

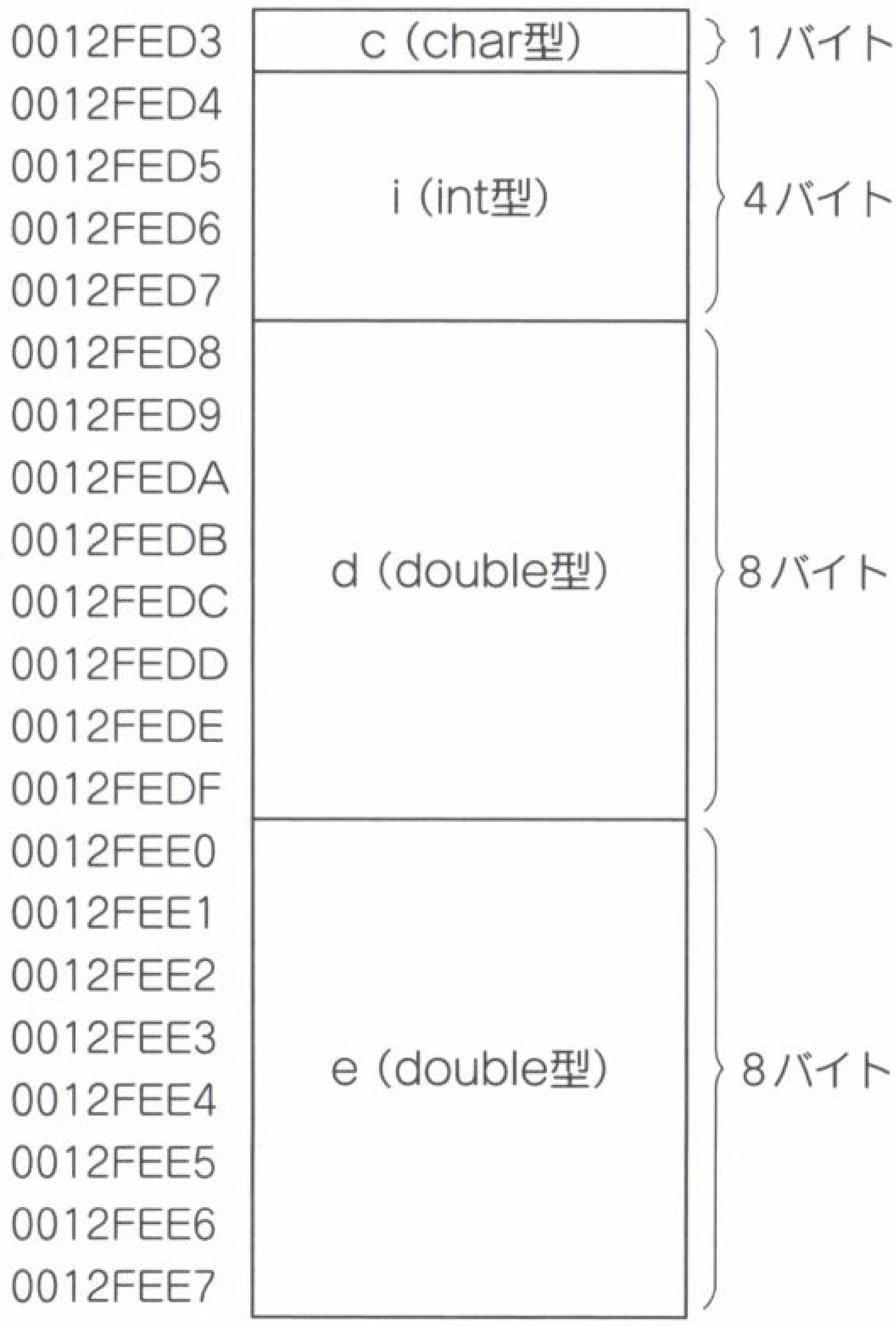


図 7-2　メモリ内の変数

double型の変数のアドレスを格納するポインタをpdとすると

```
double *pd;
```

というように宣言します。このときpdは「double型へのポインタ」となります。ポインタ型を宣言するときの「*」のことを**ポインタ宣言子**(pointer declarator)といいます。

「うーーん。ちょっと変だな。変数がchar型だろうがdouble型だろうが、そのアドレスは全部32ビットだ。どうして変数の型によってポインタの宣言が違うんだ！？」

はい、もう少ししたらわかります。今は、このように宣言するんだと覚えておいてください。

他の型でも同様です。ポインタの名前は好きに付けてよいのですが、一般の変数と区別するためにp(ponter)とかlp(long pointer)で始まる名前を付け

ることが多いようです。これだと一目でポインタだとわかりますね。

さて、

```
int a;
int *pa;
pa = &a;
```

とすると、変数aのアドレスがpaに代入されましたね。このとき、「ポインタpaは変数aを指している」とか「変数aへのポインタpa」などと表現します。

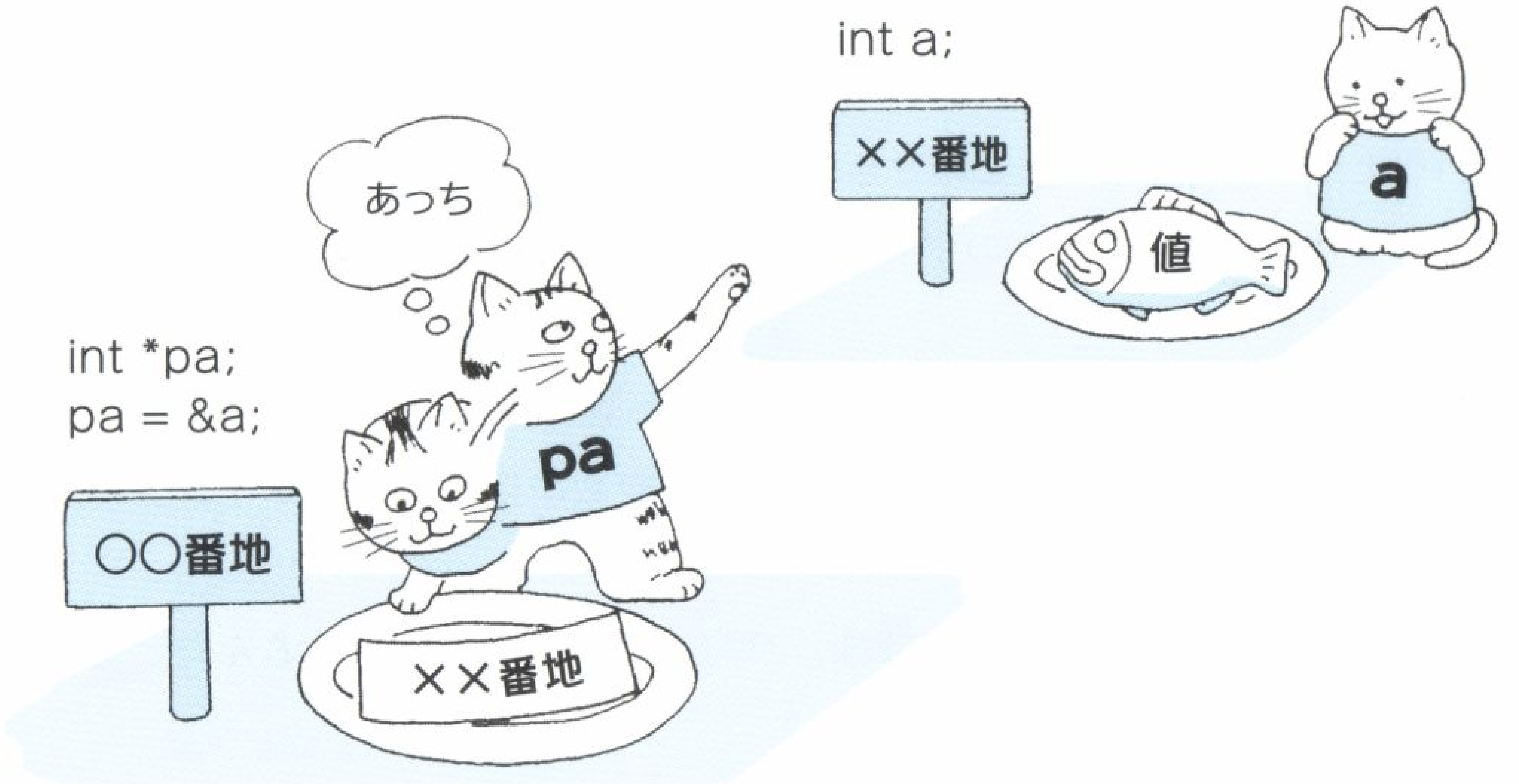
次に変数aに値を代入してみましょう。

```
int a, b;
int *pa;

a = 5;      /* 変数aに5を代入 */
pa = &a;    /* ポインタpaにaのアドレスを代入 */
b = *pa;    /* bに5が代入される！ */
```

このとき「*pa」は変数aの値を表しています。「*」のことを**間接参照演算子**(indirection reference operator)といいます。

これで前ページの疑問が解決したでしょうか。もし、ポインタを全部いっし



paにaのアドレスが代入されている場合、
paの値を見れば、aの番地がわかる

ょくたにしてしまうと、間接参照演算子を使ったときに困るわけです。paがint型へのポインタであるとわかれば、*paはint型であるということがわかりますね。

ポインタの宣言について、ちょっと考えてみましょう。

```
int *pa;
```

について*paをXで表すとします。そうすると、

```
int X;
```

となります。これは見慣れた形ですね。さらに、これを見るとXはint型であるとわかります。つまり、*paはint型ということも言えます。ポインタを使った簡単なプログラムを作ってみましょう。


```
/* pointer01.c */

#include <stdio.h>

int main()
{
    int a;
    int *lpa;

    lpa = &a;

    printf("適当な整数値を入力してください----");
    scanf("%d", &a);
    printf("変数aに%dが代入されました。¥n", a);
    printf("変数aのアドレスは%pです。¥n", &a);
    printf("変数aを指しているポインタはlpaです。¥n");
    printf("*lpaの値は%dです。¥n", *lpa);

    return 0;
}
```

ポインタの宣言

ポインタにアドレスを代入

実行結果は次のようになります。

```
適当な整数値を入力してください----10
変数aに10が代入されました。
変数aのアドレスは0012FEE4です。
変数aを指しているポインタはlpaです。
*lpaの値は10です。
```

さて、このプログラムを見て何か気がつくことがありませんか？

「そういえばscanf関数を使うとき、変数名に&をつけていたな……。これってscanf関数に変数のアドレスを教えていたのかな？？」

そうなんです！　C言語では関数の引数に変数を渡しても、受け取った関数側では変数の値しかわかりません。関数を呼び出した側の変数には何の変化も起こらないのです。これは、不用意に変数の値が変わってしまうのを防ぐ安全装置とも考えられます。

しかしscanf関数には、変数aにユーザーの入力した値を入れてほしいですね。こういう場合には、変数のアドレスを教えてやる必要があるのです。

重要

ポインタとは、変数のアドレスを格納するための変数である。

7.3 ポインタへの値代入

次の疑問がわきます。

```
int a;
int *pa;
```

としたとき、*paをint型の変数と考えて、値を代入できないだろうか？　という疑問です。答えは半分Yesで半分Noです。paが指し示している変数がわ

かっているときはYesです。しかし宣言しただけの場合は、paがメモリのどのアドレスを指しているか不明です。変数を入れる場所がわからないので、Noとなります。疑問に思ったら、自分でプログラムを作って試してみましょう。

```
/* pointer02.c */

#include <stdio.h>

int main()
{
    int *pa;
    int a;

    pa = &a;
    *pa = 5;

    printf("*pa = %d¥n", *pa);
    printf("a = %d¥n", a);

    return 0;
}
```

（int *pa; の注記）paは不定（ゴミが入っている）

（pa = &a; の注記）変数aのアドレスが代入された

（*pa = 5; の注記）aのアドレスに5を代入、つまりaに5を代入

実行結果は次のようになります。

```
*pa = 5
a = 5
```

これは、うまくいきますね。*paに5を代入したら、結果としてaに5を代入したのと同じになりました。

```
int a;
```

と宣言すると、メモリ上のどこかに4バイト分（int型を4バイトとすると）の領域が確保されます。aのアドレスを100番地とすると、100番地から103

番地までの４バイトがaのために使えるわけです（図7-3）。paにaのアドレスを代入するということは、paに100を代入するということです。

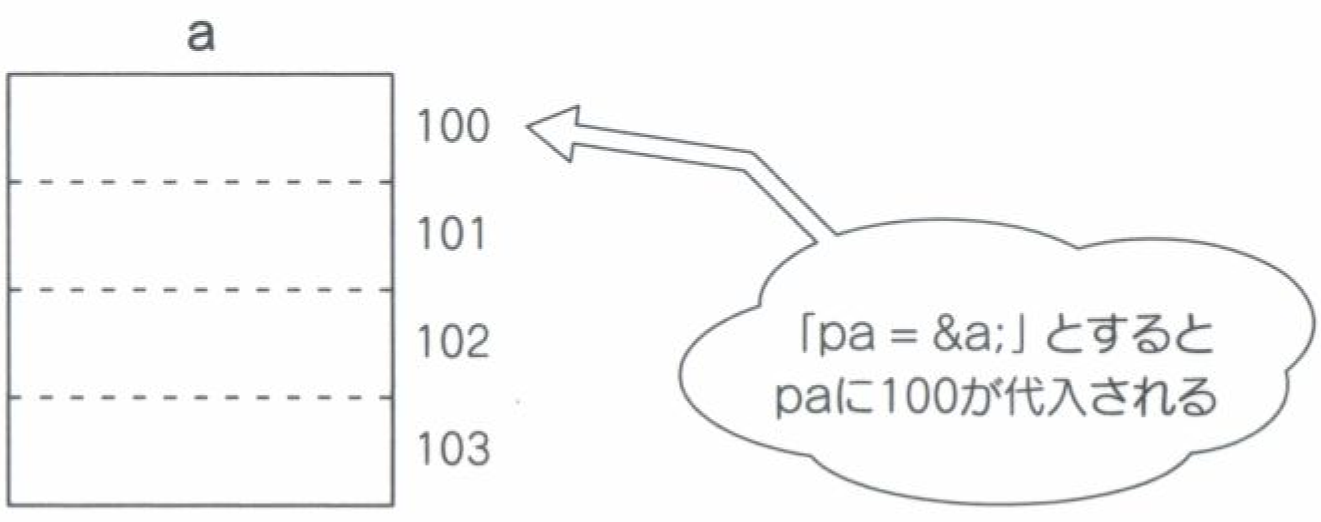


図7-3　ポインタにアドレスを代入

*paに５を代入するということは、100番地から103番地の４バイトの入れ物に５を代入するということです。なぜ100番地から４バイト分とわかるかというと、paがint型へのポインタだからです。結局、*paに５を代入するということは、aに５を代入するということと同じなのです。

さて、ポインタは変数ですから、一度何かのアドレスを入れた後、別なアドレスに変更することも可能です。また、一般の変数と同じように、宣言時にアドレスを代入して初期化することも可能です。

また、一度に複数のポインタを宣言するとき、

```
int *a, *b, *c;
```

というように書くことができます。ただし、次のように書くと、

```
int *a, b, c;
```

aのみがint型へのポインタで、bやcは普通のint型の変数となります。

```
/* pointer03.c */

#include <stdio.h>

int main()
{
    int a, b;
```

```
008:     int *p = &a;
009:
010:     a = 5;
011:     b = 10;
012:
013:     printf("a = %d, b = %d, *p = %d¥n", a, b, *p);
014:
015:     p = &b;
016:
017:     printf("a = %d, b = %d, *p = %d¥n", a, b, *p);
018:
019:     return 0;
020: }
```


実行結果を予想してみてください。もうわかりますね。最初pは変数aを指していましたが、途中(15行目)から変数bを指すように変更されました。実行結果は次のようになります。

```
a = 5, b = 10, *p = 5
a = 5, b = 10, *p = 10
```

7.4 ポインタと引数

次のプログラムは有名な失敗プログラムです。変数aと変数bの値を入れ替えるためのswap関数を作ってみましたが……。

```
/* swap01.c（意図した動作をしないプログラム）*/

#include <stdio.h>

```

```
005: void swap(int, int);
006:
007: int main()
008: {
009:     int a, b;
010:
011:     a = 10;
012:     b = 20;
013:
014:     swap(a, b);
015:
016:     printf("a = %d, b = %d¥n", a, b);
017:
018:     return 0;
019: }
020:
021: void swap(int a, int b)
022: {
023:     int c;
024:
025:     c = b;
026:     b = a;
027:     a = c;
028:
029:     return;
030: }
```

swap関数でaとbの値を入れ替えてもらうことを期待（014行目）

bの値をcに代入（025行目）

aの値をbに代入（026行目）

cに入れてあった元のbの値をaに代入
これで、aとbの値が入れ替わったはずだが…（027行目）

実行結果はどうでしょう。

```
a = 10, b = 20
```

swap関数で変数aと変数bの値を入れ替えようと思ったのですが、11～12行目で代入した「a=10」、「b=20」のままとなっています。見事に失敗ですね。

Column

初期化されていないポインタ

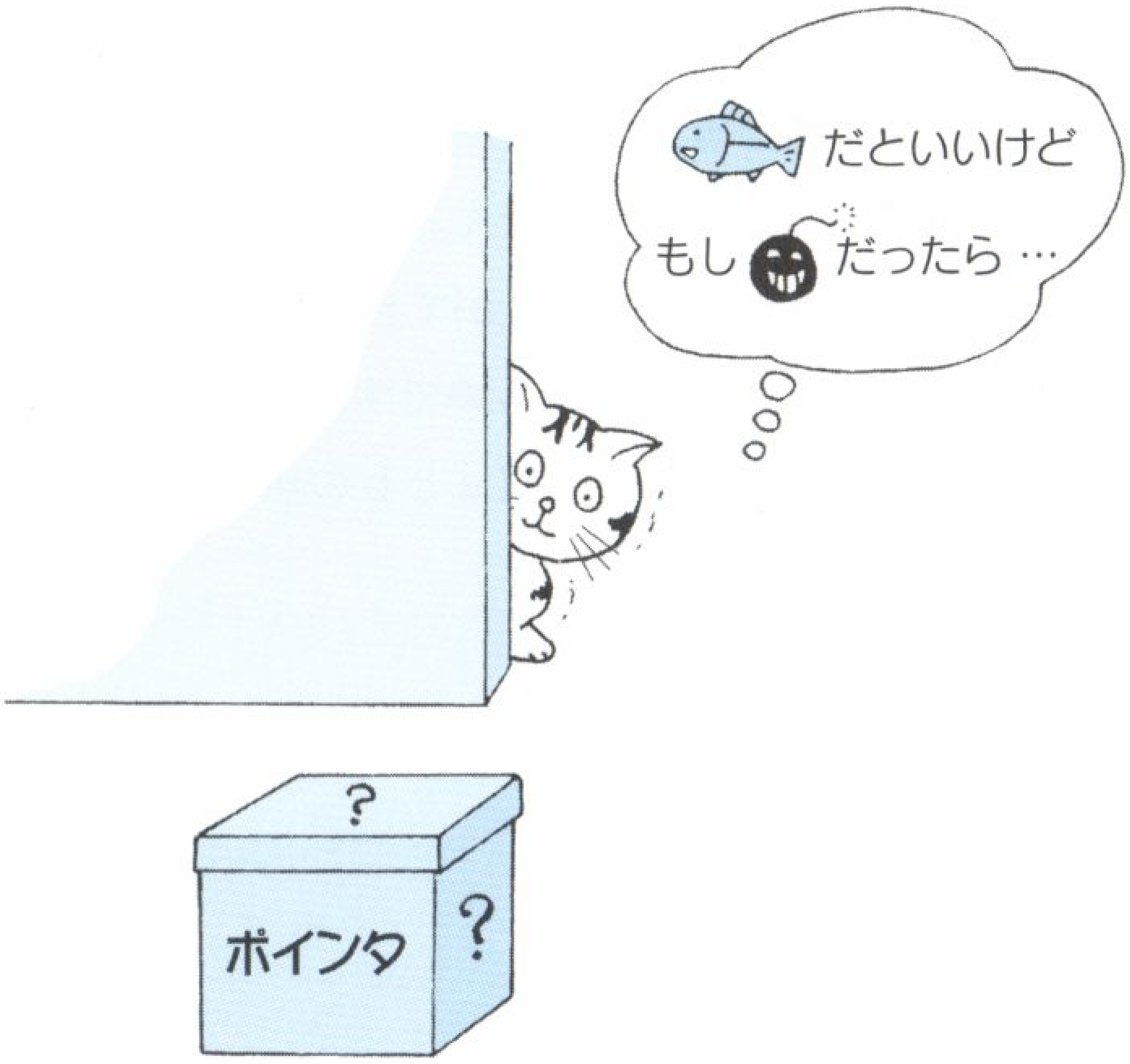
先の pointer02.c で

```
pa = &a;
```

の部分を取り除いたらどうでしょうか。コンパイル時に警告が出たのではないでしょうか。コンパイラによっては、エラーになるかもしれません。しかし、警告が出ただけで exe が作られてしまった場合、これを実行してはいけません。

pa は宣言しただけなので、中身は何が入っているかわかりません。

このでたらめなアドレスに対して 5 を代入するということは、非常に危険な行為です。最悪の場合、システムが壊れてしまうかもしれません。



初期化されていないポインタには
何が入っているかわからない

呼び出し側（main関数）の14行目で、

```
swap(a, b);
```

としていますが、これは変数aと変数bの値そのものをswap関数に渡しているだけです。swap関数では与えられた値を使って入れ替える作業をしていますが、swap関数内の変数とmain関数の変数は別物です。swap関数内で扱った変数値は、29行目のreturnの時点でどこにも保存されておらず、main関数の変数には影響を与えません。だからmain関数の変数aと変数bにはなんの変化もなかったわけです。

このような関数呼び出しを**値呼び出し**（call by value）といいます。これは一種の安全装置とも考えられます。関数呼び出しによって不用意に変数の値が変えられることを防いでいるわけです。

これに対して、関数内で仮引数の値を変更すれば呼び出し元の実引数も変更される、という方法を**参照呼び出し**（call by reference）といいます。

実はC言語には参照呼び出しがありません。しかし引数に変数のアドレスを渡すことにより、参照呼び出しと同じことができます。アドレスを渡せば呼び出し元の変数の値を変更することができるのです。

それではswap01.cを書き直して、変数aとbの値を交換してみましょう。

```
/* swap02.c */

#include <stdio.h>

void swap(int *, int *);

int main()
{
    int a, b;

    a = 10;
    b = 20;

    /* swap関数に変数a, bのアドレスを教える */
```

```
015:     swap(&a, &b);
016:
017:     printf("a = %d, b = %d¥n", a, b);
018:
019:     return 0;
020: }
021:
022: void swap(int *x, int *y)
023: {
024:     int z;
025:
026:     z = *y;
027:     *y = *x;
028:     *x = z;
029:
030:     return;
031: }
```


では、実行してみましょう。

```
a = 20, b = 10
```

11～12行目で変数a、bに代入した値が、今度は入れ替わっていますね。15行目で、変数a、bのアドレスをswap関数に渡しています。変数名に「&」がついていますね。呼び出されるswap関数側の仮引数はポインタ変数になっています(22行目)。ポインタはアドレスを入れる変数でしたね。アドレスを渡すことで、呼び出し元の変数の実体を書き換えることが可能になります。

重要

変数の値を関数に変更してもらうには、変数のアドレスを渡す。

7.5 ポインタのポインタ

ポインタ自体も変数です。ですからメモリ内のどこかに置かれているはずです。そのポインタ変数が格納されているアドレスを指すポインタ変数（つまりポインタのポインタ）を作ることはできるのでしょうか。はい、できます。次の例を見てください。

```
001: /* pointer04.c */
002:
003: #include <stdio.h>
004:
005: int main()
006: {
007:     int a;
008:     int *p;
009:     int **pp;
010:
011:     p = &a;
012:     pp = &p;
013:
014:     **pp = 10;
015:
016:     printf("a = %d, *p = %d, **pp = %d¥n", a, *p, **pp);
017:
018:     return 0;
019: }
```

11行目でポインタpに変数aの値を代入しています。そして12行目で、ポインタpの値をppに代入しています（図7-4）。さて、「**pp」はどのように解釈したらよいのでしょうか。

これは、間接参照演算子（*）が2つ並んでいると考えましょう。優先順位が同じ演算子が2つ並んでいた場合、今度は結合規則が適用されます。

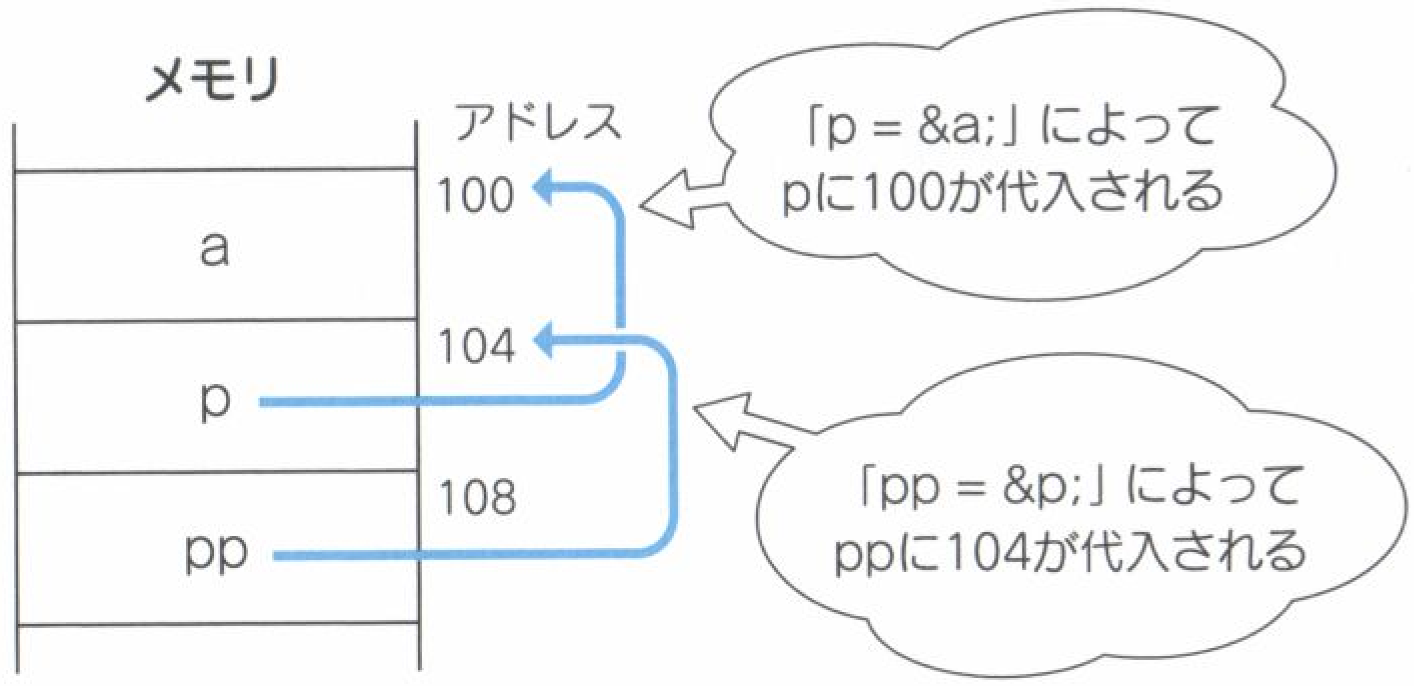


図 7-4　**ポインタのポインタ**

間接参照演算子の結合規則は←（右結合）ですね（62 ページの表を見てください）。ということは、右側の * が優先されます。つまり「**pp」は

```
*(*pp)
```

ということになります。「*pp」は p の値（a のアドレス）です。ということは

```
*(p)
```

と同じです。結局、**pp に 10 を代入するということは、a に 10 を代入するということと同じになるわけです。

では、実行結果を見てみましょう。

Column

NULL

ヌルポインタは、NULL と書き、特別なポインタを意味しています。文字通り解釈すると 0 番地を指しているポインタのことです。

C 言語では、0 番地は使わない約束になっており、もしポインタに 0 が格納されていたら、これは不正なポインタであることを意味します。

関数の戻り値（134 ページ参照）としてポインタが返されることがよくありますが、関数が失敗したときなどに NULL を返すものがたくさんあります。したがってポインタを使うプログラムでは、使うポインタが NULL になっていないことをよく確かめる必要があります。

```
a = 10, *p = 10, **pp = 10
```

どうでしょう、少しはポインタに親しみが持てるようになりましたか？

練習問題

1. カッコ内に適当な語句や記号を入れて、文章を完成させてください。

a. 変数aのアドレスは（　1　）演算子を使って（　2　）と表すことができる。
b. ポインタには（　3　）の（　4　）を代入することができる。
c. 次のコードを実行すると、*ptrの値は（　5　）となる。

```
int a = 5;
int *ptr;
ptr = &a;
```

※解答は付録CD-ROMに収録

第7章のまとめ

1 アドレスはメモリ上の位置を表す。

2 変数に&（アドレス演算子）つけると、変数のアドレスを表す。

3 ポインタは、アドレスを格納するための変数である。

4 ポインタに間接参照演算子「*」を使うと、ポインタが指している変数の値を取得できる。

5 関数で、呼び出し元の変数の値を直接変更するには、変数のアドレスを渡す。

6 ポインタのポインタも使うことができる。

第8章 配列・文字列とポインタ

配列と文字列とポインタはそれぞれ切っても切れない関係にあります。この章では、これらの関係を順を追って解説していきます。

8.1 配列の基本

40人いる生徒の100メートル走のタイムを記録して、平均値を出すプログラムを考えてみましょう。平均値を出すだけではなく、1人1人の記録も呼び出せるようにしたい、とします。さて、どうしましょう。40人分の変数を用意するとなると、面倒です。

```
double a, b, c, ....;
```

というようなことをしていては日が暮れてしまいますね。こんなとき、**配列** (array) が役立ちます。配列は次のように宣言します。

```
データ型 配列名[配列の大きさ];
```

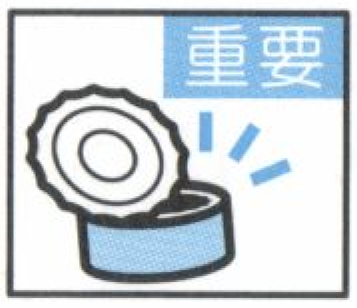
たとえば

```
int a[10];
```

というような感じで宣言します。このように宣言すると「a[0], a[1], … a[9]」をそれぞれint型の変数として取り扱うことができます。これを配列の**要素**といいます。

[ ]の中の0, 1, … は配列の**添え字**(index)といいます。添え字は必ず0から始まり1ずつ増えていきます。途中で数字が飛んだりはしません。配列の大きさとは要素の数なので、添え字は0から(配列の大きさ−1)まであります。

上のように宣言した場合a[10]は存在しませんが、a[10]を間違って使ってもコンパイラは警告を出したり、エラーにしたりはしません。間違ってa[10]を使ったプログラムを動作させた場合、何が起こるかわかりません。



- 配列は変数の集まりである。
- 配列の添え字は、0～(配列の大きさ−1)である。

さて、次に重要なことはa[0], a[1], … はこの順番でメモリ上に隙間なく並んでいる、という点です。int型を4バイトとし、a[0]のアドレスが100番地であるとすれば、a[1]のアドレスは104番地、a[2]は108番地、....ということになります(図8-1)。これは、非常に重要なことです。

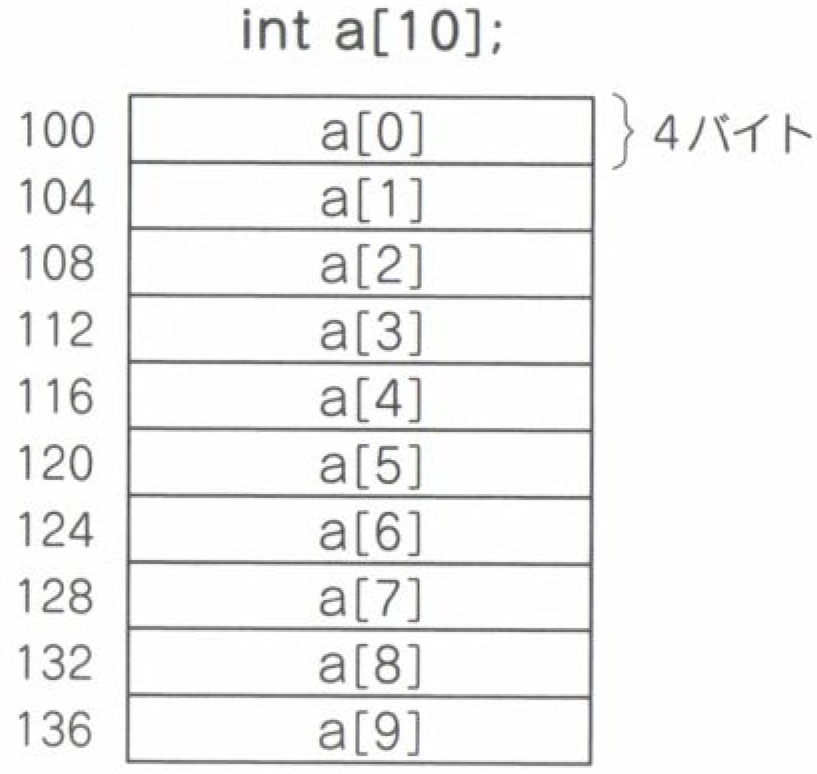


図8-1　配列のメモリ内の状態

では、まずは簡単なサンプルプログラムを見てみましょう。

```
/* array01.c */

#include <stdio.h>

int main()
{
    int a[4];

    a[0] = 1;
    a[1] = 2;
    a[2] = 100;
    a[3] = 50;

    printf("a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d¥n",
        a[0], a[1], a[2], a[3]);

    return 0;
}
```

配列の宣言

これは説明するまでもないですね。実行結果は次のようになります。

```
a[0] = 1, a[1] = 2, a[2] = 100, a[3] = 50
```

このプログラムのように配列の要素に値を代入するのは、いかにも面倒くさいですね。何かいい方法はないでしょうか。あるんですね、これが。

```
int a[4] = {1, 2, 100, 50};
```

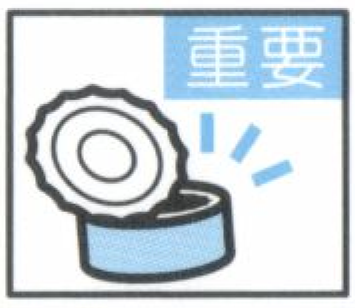
と書けば、配列の宣言と同時に各要素に値が代入され初期化されます。また、要素の数を省略して

```
int a[] = {1, 2, 100, 50};
```

のように書くこともできます。この場合自動的に要素の数は４となります。また、

```
int a[4] = {1, 2};
```

とした場合、a[0]は１，a[1]は２となるのはわかりますが、a[2]とかa[3]はどうなっているのでしょうか。実はa[2]，a[3]には０が代入されます。



配列は宣言と同時に初期化することができる。

では、array01.cをもう少し簡潔に書き直してみましょう。

```
001: /* array02.c */
002:
003: #include <stdio.h>
004:
005: int main()
006: {
007:     int a[] = {1, 2, 100, 50};
008:     int i;
009:
010:     for (i = 0; i < 4; i++)
011:         printf("a[%d] = %d, ", i, a[i]);
012:     printf("¥b¥b ¥b¥n");
013:
014:     return 0;
015: }
```

12行目の意味がわかるでしょうか。12行目がないと

```
a[0] = 1, a[1] = 2, a[2] = 100, a[3] = 50,
```

のように最後にコンマとスペースがついてしまいます。そこでバックスペース(¥b)で２つ戻りスペースを出力して最後のコンマを消します。その後バックスペースで１つ戻り、今出力したスペースのところに改行を出力しています。

変数iの使い方にも注目してください。添え字には上の例のように変数を使えるのでたいへん便利です。

さて、配列の初期化で次のようにしたらどうでしょうか。

```
int a[2] = {1, 2, 3};
```

これは、「初期化子が多すぎる」というコンパイルエラーになります。

では、これはどうでしょうか。2の後ろのコンマは不要なのでエラーになるのでしょうか。

```
int a[] = {1, 2, };
```

いいえ。これは、間違いではないのです。このような書き方もあります。後から出てきますが文字列の配列の場合はよくこのような書き方をします。

8.2 配列とアドレス

配列の要素は原則として、順番にメモリ上に並んでいます(168ページ参照)。それでは実際に各要素のアドレスを調べるプログラムを作ってみましょう。

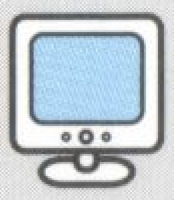
```
/* array03.c */

#include <stdio.h>

int main()
{
    int a[4] = {1, 2, 3, 4}, i;

    for (i = 0; i < 4; i++) {
        printf("&a[%d] = %p¥n", i, &a[i]);
    }
    printf("¥na = %p¥n", a);
    return 0;
}
```

さっそく実行結果を見てみましょう。



```
&a[0] = 0012FEC8
&a[1] = 0012FECC
&a[2] = 0012FED0
&a[3] = 0012FED4

a = 0012FEC8
```

確かにa[0]からa[3]までのアドレスは4バイトずつ隙間なくつながっていますね。

「あれっ！ 最後のa＝0012FEC8って？？？」

このアドレスと、a[0]のアドレスを見比べてください。なんと同じですね！しかもaは配列の名前です。実は配列の名前は、その配列の先頭要素のアドレスを表しているのです(蛍光ペン！！)。



配列名は、その配列の先頭要素のアドレスを表している。

ということは、int型(4バイトであるとする)の配列であれば、aで示されるアドレスより4バイト進めるとa[1]のアドレスになるということです。これを確かめてみましょう。

```
/* array04.c */

#include <stdio.h>

int main()
{
    int a[] = {10, 20, 30, 40}, i, *p;

```

```
009:     p = a;     /* pにa[0]のアドレスを代入 */
010:
011:     for (i = 0; i < 4; i++) {
012:         printf("&a[%d] = %p, a[%d] = %d, *(p + %d) = %d¥n",
013:             i, &a[i], i, a[i], i, *(p + i));
014:     }
015:
016:     return 0;
017: }
```

これを実行してみます。

```
&a[0] = 0012FED8, a[0] = 10, *(p + 0) = 10
&a[1] = 0012FEDC, a[1] = 20, *(p + 1) = 20
&a[2] = 0012FEE0, a[2] = 30, *(p + 2) = 30
&a[3] = 0012FEE4, a[3] = 40, *(p + 3) = 40
```

あれ、ちょっとおかしいのではないでしょうか。aが配列の先頭要素のアドレスであることはわかりました。これをint型へのポインタpに代入しました(9行目)。a[1]はpよりも4バイト進んだ番地なので(p + 4)番地ではないでしょうか。したがってa[1]の値は*(p + 4)なのではないでしょうか。

いいえ、「int型へのポインタに1加える」ということは、「int型の大きさだけアドレスを1つ進める」ということなのです(他の型についても同様)。

さらに、配列の名前aはポインタのように(a + 1)とすると、aより4番地進んだアドレスを表します。これも確かめてみましょう。

```
/* array05.c */

#include <stdio.h>

int main()
{
    int *p, i, a[] = {20, 40, 80, 100};
```

```

    p = a;

    for (i = 0; i < 4; i++) {
        printf("&a[%d] = %p, (p + %d) = %p, (a + %d) = %p¥n",
            i, &a[i], i, p + i, i, a + i);
    }

    return 0;
}
```

これを実行すると次のようになります。

```
&a[0] = 0012FEB0, (p + 0) = 0012FEB0, (a + 0) = 0012FEB0
&a[1] = 0012FEB4, (p + 1) = 0012FEB4, (a + 1) = 0012FEB4
&a[2] = 0012FEB8, (p + 2) = 0012FEB8, (a + 2) = 0012FEB8
&a[3] = 0012FEBC, (p + 3) = 0012FEBC, (a + 3) = 0012FEBC
```

どうです。ちょっと驚きですね。また、このことよりa[n]の値は*(a + n)と同じであるといえます(nは配列の大きさ以内の数)。

「配列もポインタも同じなのか…」

確かに似た性質があります。しかしまったく同じではありません。ポインタは変数なので中身を書き換えることができました。しかし、配列名はあくまでも先頭要素のアドレスを示すものであり変数ではありません。したがって書き換えることはできないのです。ポインタpに対してp++は可能ですが、配列名aに対してa++はできません。

さて、配列の[ ]も演算子なのです。62ページの表4-1を見てください。ちゃんと載っていますね。これを**配列添え字演算子**(array subscription operator)といいます(配列を宣言するときの[ ]は**配列宣言子**(array declarator)といいます)。

さて、a[n]は、*(a + n)と同じであると書きました。*(a + n)と*(n + a)は同じなのでしょうか。言い換えると「a + n」と「n + a」は同じなのでしょうか。

実は同じなのです。プログラムを書いて確かめてみてください。ということは、*(a + n)がa[n]と同じなので、*(n + a)はn[a]と同じ。つまりa[n]はn[a]と書いても同じことなのです！

a[5]を5[a]と書いても同じ意味ですが、後者のような使い方をしているとわけがわからなくなりますのでお勧めしません。

今までの議論から、配列添え字演算子はポインタに関しても適用できそうな気がしますね。結論的にいうと使えます！！ 次の３つは同じなのです。配列とポインタには密接な関係があるというのがわかってきたでしょうか。

```
&str[n]
pstr + n
&pstr[n]
```


8.3 多次元配列

8.1節で解説した配列は、要素が１つのみでした。これを１次元配列といいます。これに対して、次のような配列があります。

```
int a[5][2];
```

イメージ的には５×２のマス目を考えるとわかりやすいです。

さて、この配列がメモリ上でどのように並んでいるかというと、

```
a[0][0]
a[0][1]
a[1][0]
a[1][1]
a[2][0]
a[2][1]
a[3][0]
a[3][1]
a[4][0]
a[4][1]
```

の順に隙間なく並んでいます。また、1次元配列と同様にa(配列の名前)は、a[0][0]のアドレスを表しています。

配列の初期化は

```
int a[5][2] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

のように書きます。また、

```
int a[][2] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

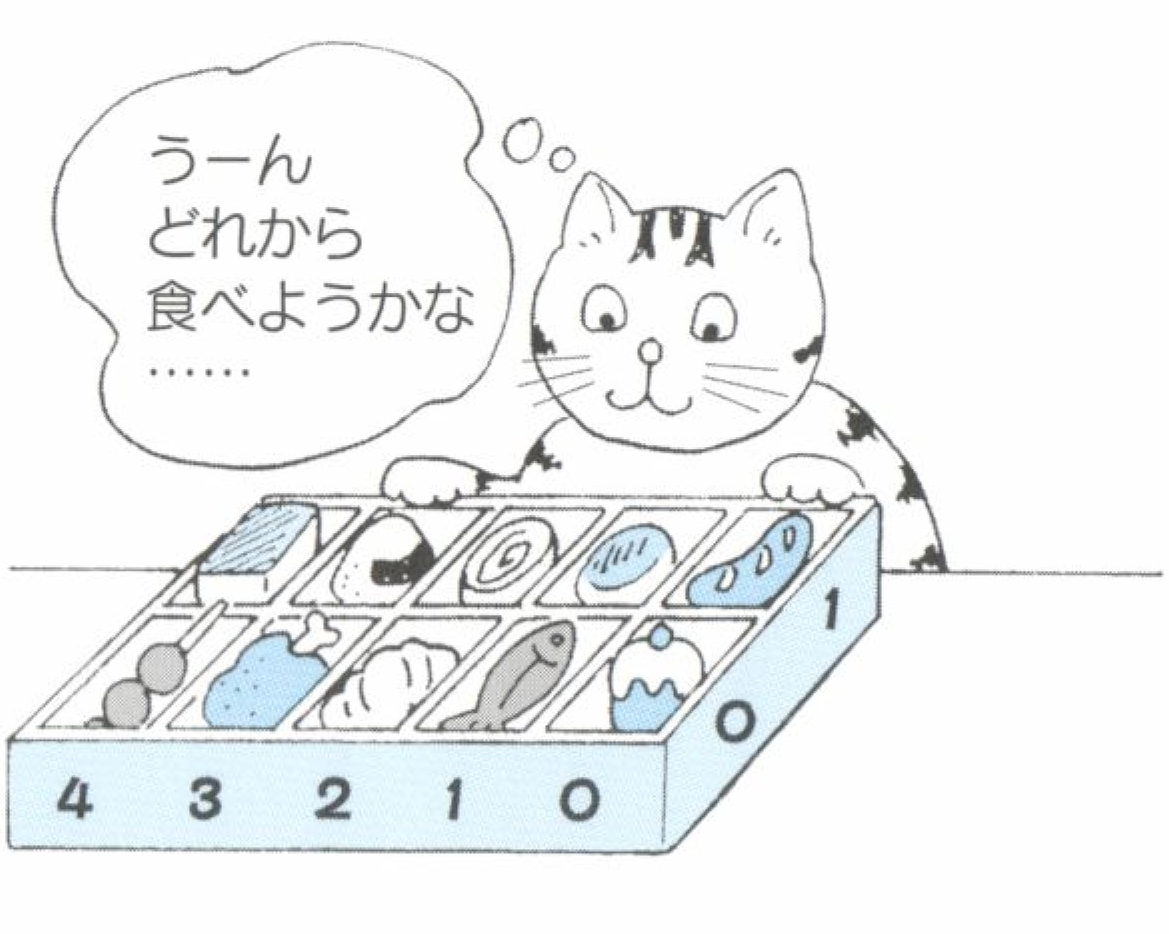
のようにも書くことが許されています(最初の要素数のみ省略可)。さて、このように書いたとき、各要素の値はわかりますか。{}の中身は、メモリに並ぶ順番で書いています。もう、わかりましたね。

同様にして

```
int a[i][j][k];
```

のように3次元配列も作ることができます。次元が2以上の配列を**多次元配列**(multi-dimensional array)といいます。メモリに並ぶ順番は、「多次元配列は、一番外側(右)から回る」と覚えましょう。3次元だろうが4次元だろうが同じです。初期化の方法も2次元配列と同じです。

さて、この配列でa[2][0]について考えてみましょう。配列の[ ]は演算子でした。1つの式の中に2つのの演算子があることになります。しかも、この2つは同じ演算子なので優先順位は等しいです。となると、結合規則が適用されます。[ ]の結合規則は→ですね。ということは



```
(a[2])[0]
```

と書いても同じです。a[2]をAで置き換えると

```
A[0]
```

となり、一次元配列のように見えます。するとAが配列名で、これはA[0]のアドレスを表しているのですね。

また、A[0]は*(A + 0)とも書けました。これを元に戻すと次のようになります。

```
*(a[2] + 0)
```

また、a[2]は*(a + 2)と書けました。したがってa[2][0]は、

```
*(*(a + 2) + 0)
```

と書けることがわかります。ちょっとこのあたりのことを確かめるプログラムを作ってみましょう。

a[i][j]のアドレスは&a[i][j]と書くことができます。これは、&と[ ]の演算子では[ ]のほうが優先順位が高いので、&(a[i][j])と同じです。かっこの中は同じ演算子([ ])どうしなので結合規則が適用されることはすでに述べました。

```
/* array06.c */

#include <stdio.h>

int main()
{
    int a[3][2] = {10, 20, 30, 40, 50, 60};
    int i, j;

    /* 各要素の値を確かめる */
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 2; j++) {
            printf("a[%d][%d] = %d¥n", i, j, a[i][j]);
        }
    }
```

```

    printf("¥n");

    /* 各要素のアドレスを確かめる */
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 2; j++) {
            printf("&a[%d][%d] = %p   (a[%d] + %d) = %p¥n",
                i, j, &a[i][j], i, j, (a[i] + j));
        }
    }

    printf("¥n");

    /* a[m][n]を*(*(a + m) + n)と書けるのか確かめる */

    for (i = 0; i < 3; i++) {
        for (j = 0; j < 2; j++) {
            printf("*(*(a + %d) + %d) = %d¥n",
                i, j, *(*(a + i) + j));
        }
    }

    return 0;
}
```

では、実行結果を見てみましょう。

```
a[0][0] = 10
a[0][1] = 20
a[1][0] = 30
a[1][1] = 40
a[2][0] = 50
a[2][1] = 60

&a[0][0] = 0012FEC0   (a[0] + 0) = 0012FEC0
&a[0][1] = 0012FEC4   (a[0] + 1) = 0012FEC4
&a[1][0] = 0012FEC8   (a[1] + 0) = 0012FEC8
```

```
&a[1][1] = 0012FECC   (a[1] + 1) = 0012FECC
&a[2][0] = 0012FED0   (a[2] + 0) = 0012FED0
&a[2][1] = 0012FED4   (a[2] + 1) = 0012FED4

*(*(a + 0) + 0) = 10
*(*(a + 0) + 1) = 20
*(*(a + 1) + 0) = 30
*(*(a + 1) + 1) = 40
*(*(a + 2) + 0) = 50
*(*(a + 2) + 1) = 60
```

予想通りの結果ですね。

では、もう少し実用的なプログラムを作ってみましょう。生徒５人について、英語、数学のテストを行いました。各教科の平均点を出し、また個人別総合得点、および平均点を出すプログラムを作ってみましょう。

その前にちょっと補足しておきますが、プログラムの冒頭に

```
#define NO 3
```

という行があります。これは「NO」は「3」であると定義しています。つまりプログラム中に「NO」が出てくると、これを「3」に置き換えます。たとえば「sum / NO;」は「sum / 3;」となります。次のプログラムでは生徒の数は３で、この３が繰り返し出てきます。これをNOで置き換えると間違いを起こしにくくなります。

```
/* array07.c */

#define NO 3  /* NOを３と定義 */

#include <stdio.h>

int main()
{
    int point[][2] = {
          80, 80,     /* 出席番号１の英語、数学の得点 */
```

```
        100, 98,     /* 出席番号 2 */
         60, 80,     /* 出席番号 3 */
    };

    int i, j, sum = 0, p_sum[NO];
    double ave;

    /* 英語の平均点を求める */
    for (i = 0; i < NO; i++)
        sum += point[i][0];
    ave = (double)sum / NO;

    printf("英語の平均点は %5.1f 点です¥n", ave);

    /* 数学の平均点を求める */
    sum = 0; /* 合計点を 0 にもどしておく */
    for (i = 0; i < NO; i++)
        sum += point[i][1];
    ave = (double)sum / NO;

    printf("数学の平均点は %5.1f 点です¥n¥n", ave);

    /* 個人別合計点を求める */
    for (i = 0; i < NO; i++)
        p_sum[i] = 0;

    for (i = 0; i < NO; i++) {
        for (j = 0; j < 2; j++) {
            p_sum[i] += point[i][j];
        }
        printf("出席番号 %d の総得点 %d¥n", i + 1, p_sum[i]);
    }

    return 0;
}
```

実行結果は次のようになります。

```
英語の平均点は 80.0点です
数学の平均点は 86.0点です

出席番号 1の総得点 160
出席番号 2の総得点 198
出席番号 3の総得点 140
```

配列を使うと、このように表計算のようなプログラムを簡潔に書くことができます。

8.4 文字と文字列

文字は、'A', 'B' などのようにシングルクォーテーションで囲んで表現しました。また、printf関数で文字出力するときの変換指定子は%cでした。まずは簡単なプログラムから。

```c
/* char01.c */

#include <stdio.h>

int main()
{
    char a1, a2, a3;

    a1 = 'a';
    a2 = 'b';
    a3 = 'c';

    printf("%c%c%c¥n", a1, a2, a3);

    return 0;
}
```

実行結果は説明するまでもなく

```
abc
```

ですね。では、次はどうなるでしょうか。

```
/* char02.c */

#include <stdio.h>

int main()
{
    char a1, a2, a3;

    a1 = 'A';
    a2 = 'B';
    a3 = 'C';

    printf("a1 = %d, a2 = %d, a3 = %d¥n", a1, a2, a3);

    return 0;
}
```

「あれ、このプログラム間違っているぞ！！ a1，a2，a3はchar型なので%cじゃないの？」

しかし、コンパイラは何の文句もいわずにビルドしましたね。では、実行結果を見てみましょう。

```
a1 = 65, a2 = 66, a3 = 67
```

実は 'A' とか 'B' の文字は、その文字のASCIIコードを表しているのです。要するに単なる数字なのですね（ASCIIコード表については付録C参照）。

char型は1バイトのデータ型でしたね。多くのコンパイラでは、単にchar型といえばsigned char型と同じ意味に解釈され、扱えるデータは–128から127までの整数値です。しかしchar型の符号の有無については規定がなく、コンパイラによってはunsigned charの意味で使われることもあります。

はっきりさせないと困る場合はsignedとかunsignedをつけて明示的に使う必要があります。

また、文字は単なる数字なので、これをint型で取り扱っても問題はありません。次の例では文字をfor文の中で使っています。文字は単なる数字なのでこのような使い方ができるのです。

```
/* char03.c */

#include <stdio.h>

int main()
{
    char i;

    for (i = '!'; i <= '~'; i++) {
        printf("%3d(0x%2X) -- %c  ", i, i, i);
        if ((i - '!' + 1) % 4 == 0)
            printf("¥n");
    }
    printf("¥n");

    return 0;
}
```

さて、このプログラムはいったい何をしているのでしょうか。ASCIIコード表を眺めてみると、0番（0x00）から31番（0x1F）までは、目に見えない制御コードになっています。32番（16進で0x20）はスペースですが、これまた画面では見えませんね。

33番(0x21)から目に見える文字('!')が割り当てられています。これが126番(0x7E)('~')まで続きます。127番(0x7F)はまた制御コードとなり、目に見えません。

上のプログラムのfor文のところは

```
for (i = 33; i <= 126; i++) {
    ...
}
```

と書き直しても同じことですね。しかし、このような書き方ではいちいちASCIIコード表を調べなくてはならず、面倒です。アルファベットの 'A' から 'Z' を表示したいのなら

```
for (i = 'A'; i <= 'Z'; i++) {
    ...
}
```

という具合になるので、たいへん便利ですね。では、実行結果を見てみましょう。

```
33(0x21) -- !   34(0x22) -- "   35(0x23) -- #   36(0x24) -- $
37(0x25) -- %   38(0x26) -- &   39(0x27) -- '   40(0x28) -- (
41(0x29) -- )   42(0x2A) -- *   43(0x2B) -- +   44(0x2C) -- ,
45(0x2D) -- -   46(0x2E) -- .   47(0x2F) -- /   48(0x30) -- 0
49(0x31) -- 1   50(0x32) -- 2   51(0x33) -- 3   52(0x34) -- 4
53(0x35) -- 5   54(0x36) -- 6   55(0x37) -- 7   56(0x38) -- 8
57(0x39) -- 9   58(0x3A) -- :   59(0x3B) -- ;   60(0x3C) -- <
61(0x3D) -- =   62(0x3E) -- >   63(0x3F) -- ?   64(0x40) -- @
65(0x41) -- A   66(0x42) -- B   67(0x43) -- C   68(0x44) -- D
69(0x45) -- E   70(0x46) -- F   71(0x47) -- G   72(0x48) -- H
73(0x49) -- I   74(0x4A) -- J   75(0x4B) -- K   76(0x4C) -- L
77(0x4D) -- M   78(0x4E) -- N   79(0x4F) -- O   80(0x50) -- P
81(0x51) -- Q   82(0x52) -- R   83(0x53) -- S   84(0x54) -- T
85(0x55) -- U   86(0x56) -- V   87(0x57) -- W   88(0x58) -- X
89(0x59) -- Y   90(0x5A) -- Z   91(0x5B) -- [   92(0x5C) -- ¥
93(0x5D) -- ]   94(0x5E) -- ^   95(0x5F) -- _   96(0x60) -- `
```

```
 97(0x61) -- a   98(0x62) -- b   99(0x63) -- c  100(0x64) -- d
101(0x65) -- e  102(0x66) -- f  103(0x67) -- g  104(0x68) -- h
105(0x69) -- i  106(0x6A) -- j  107(0x6B) -- k  108(0x6C) -- l
109(0x6D) -- m  110(0x6E) -- n  111(0x6F) -- o  112(0x70) -- p
113(0x71) -- q  114(0x72) -- r  115(0x73) -- s  116(0x74) -- t
117(0x75) -- u  118(0x76) -- v  119(0x77) -- w  120(0x78) -- x
121(0x79) -- y  122(0x7A) -- z  123(0x7B) -- {  124(0x7C) -- |
125(0x7D) -- }  126(0x7E) -- ~
```

次に、文字をint型として取り扱ってみます。次のプログラムはアルファベットのASCIIコードを表示するものです。

```
001: /* char04.c */
002:
003: #include <stdio.h>
004:
005: int main()
006: {
007:     int c;
008:
009:     for (c = 'A'; c <= 'z'; c++) {
010:         if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z'))
011:             printf("%c = %3d(0x%X)¥n", c, c, c);
012:     }
013:
014:     return 0;
015: }
```

for文の中にあるif文（10行目）の意味はわかるでしょうか。ASCIIコードの 'A'～'Z' の次にアルファベット以外の文字が6文字続き、それから 'a'～'z' と続きます。アルファベット以外の文字を表示させないためのコードですね。論理演算子が複数使われています。&&と || では&&のほうが優先順位が高いので、かっこを省いて

```
if (c >= 'A' && c <= 'Z' || c >= 'a' && c <= 'z') {
    ...
}
```

と書いても問題ありませんが、これではぱっと見てすぐには理解できませんね。かっこをつけるとコードが読みやすくなります。

ABCのような文字の並びを**文字列**(string)といい、ダブルクォーテーションで囲みます。"ABC" のようなものを**文字列リテラル**(string literal)といいます。そういえば、いつも使っているprintf関数でも、

```
printf("ABC¥n");
```

というように、かっこの中にダブルクォーテーションで囲まれた部分がありますね。これも文字列なのです。printf関数の文字列の変換指定子は%sです。

```
printf("abcは%sです¥n", "アルファベット");
```

とすれば、

```
abcはアルファベットです
```

と表示されます。

さて、"ABC" がメモリに置かれる場合、'A', 'B', 'C' の順にメモリ上に置かれます。そして、'C' の次の番地には '**¥0**'(**ヌル文字**、null character)が置かれます(図8-2)。これは、文字列がここで終わりだ、という合図です。'A' が300番地に置かれたとすれば、'B' は301番地、'C' は302番地、'¥0' は303番地に置かれます。

	'A'	'B'	'C'	'¥0'
アドレス	300	301	302	303

図8-2 文字列のメモリ内の状態

そして、次が非常に重要なことですが、"ABC" の式の値は先頭文字 'A' の置かれたアドレスなのです。これは、無茶苦茶重要です。蛍光ペンでしっかりマ

ークしてください。

重要

- 文字列の式の値は、先頭文字の置かれたアドレスである。
- 文字配列の最後の要素は必ず '¥0' である。

さて、"ABC" は先頭文字のアドレスを表していますので、これを char 型へのポインタに代入することができます。

```
char *str;
str = "ABC";
```

というようなことができます。

また、ポインタを次のように初期化することもできます。

```
char *str = "ABC";
```

str はポインタなので *(str + 0)は 'A' を、*(str + 1)は 'B' を、*(str + 2)は 'C' を、*(str + 3)は '¥0' を表していますね。

また、*(str + n)は str[n]と同じでした。したがって、str[0] は 'A'、str[1]は 'B'、str[2]は 'C'、str[3]は '¥0' を表しています。では、これを確かめるプログラムを作ってみましょう。

Column

特殊記号

国により一部、同じ ASCII コードであっても表す文字が異なることがあるので注意してください。126 番は日本ではチルダ ('~') を表しますが、ドイツ、フランス、ノルウェー等では違う文字に割り当てられています。

また、92 番 (0x5C) は日本では円記号 '¥' が割り当てられていますが、アメリカ、イギリスではバックスラッシュ ('\') が割り当てられています。

```
/* string01.c */

#include <stdio.h>

int main()
{
    char *str;

    str = "ABC";

    printf("*(str + 0) = '%c', "
           "*(str + 1) = '%c', "
           "*(str + 2) = '%c'¥n",
        *(str + 0), *(str + 1), *(str + 2));
    printf("str[0] = '%c', str[1] = '%c', str[2] = '%c'¥n",
        str[0], str[1], str[2]);

    return 0;
}
```

実行結果は次のようになります。

```
*(str + 0) = 'A', *(str + 1) = 'B', *(str + 2) = 'C'
str[0] = 'A', str[1] = 'B', str[2] = 'C'
```

さて、"ABC" 自体も先頭文字のアドレスを表しているので *("ABC" + 0)は 'A' を表しているのでしょうか。また、"ABC"[0]というような書き方は許されるのでしょうか。確かめてみましょう。

```
/* string02.c */

#include <stdio.h>

int main()
```

```
{
    printf("%c, %c, %c¥n",
        *("ABC" + 0), *("ABC" + 1), *("ABC" + 2));
    printf("%c, %c, %c¥n", "ABC"[0], "ABC"[1], "ABC"[2]);

    return 0;
}
```

問題なくコンパイルできましたね。実行結果は次のようになります。

```
A, B, C
A, B, C
```

さて、次のような疑問がわきませんか。

```
char *str1, *str2;

str1 = "ABC";
str2 = "ABC";
```

str1とstr2には、同じアドレスが格納されているのだろうか?????(?がたくさんついてるので、強い疑問です)

答えは、「異なるアドレスが格納されています」です。文字列は出てくるたびにそれぞれ別々のところに格納されます。string02.cでは "ABC" が6箇所に出てきています。これらは、すべて別々なところに格納されています。

しかし、しかしです!!! 「文字列としては同じなので、別々のアドレスに格納するのは、メモリの無駄遣いだ」との考えで、文字列として同じものが出てきたら、別のアドレスに格納しないコンパイラオプションもあります。

しかし、文字列の比較という意味で

```
if (str1 == str2) {
    ...
}
```

というようなプログラムは間違いであると考えてください(たまたま、意図した目的を達成できることもありますが)。文字列の内容を比較するにはstrcmp関数を使います。

strcmp 関数

2つの文字列を比較します。

```
int strcmp(
    const char *string1,
    const char *string2
);
```

この関数を使いたいときは、string.hをインクルードする必要があります。

比較したい文字列へのポインタを引数として2つ与えます。strcmp関数は2つの文字列を辞書式に比較します。辞書式に並べたときにstring1のほうが前のほうにあればマイナス、後ろのほうにあればプラス、まったく同じなら0の値を返します。

```
/* string03.c */

#include <stdio.h>
#include <string.h>

int main()
{
    char str1[32], str2[32];
    int cmp;

    printf("str1を入力してください--");
    scanf("%s", str1);
    printf("str2を入力してください--");
    scanf("%s", str2);

    cmp = strcmp(str1, str2);
    if (cmp < 0)
        printf("%sは、%sより前にあります¥n", str1, str2);
```

strcmp関数を使うので必要（#include <string.h>）

```
    else if (cmp > 0)
        printf("%sは、%sより後ろにあります¥n", str1, str2);
    else
        printf("%sと%sは、同じ文字列です¥n", str1, str2);

    return 0;
}
```

さて、"ABC"の場合、3文字ですね。文字列の長さを調べる関数というものはないのでしょうか。あります。

strlen関数

文字列の長さを返します。

```
size_t strlen(
    const char *string
);
```

これで返される値には、ヌル文字は含まれていません。size_t型というのは、4.5節「sizeof演算子」でやりましたね。

```
/* string04.c */

#include <string.h>
#include <stdio.h>

int main()
{
    char str[32];
    size_t len;

    printf("文字列を入力してください--");
    scanf("%s", str);

    len = strlen(str);

```

```
    printf("%sの長さ%dです¥n", str, len);

    return 0;
}
```

実行結果は次のようになります。

```
文字列を入力してください--abcde
abcdeの長さ5です
```

8.5 文字配列

文字列をchar型の配列で取り扱うことはできないでしょうか。答えは「できます」。

```
char str[4];
```

という配列を考えます。これに、

```
str[0] = 'A';
str[1] = 'B';
str[2] = 'C';
str[3] = '¥0';
```

というように、各要素に文字を代入してやればよいですね。いちいち、このように代入していては面倒なので、次のように初期化することもできます。

```
char str[4] = {'A', 'B', 'C', '¥0'};
```

さらに、添え字を省略して

```
char str[] = {'A', 'B', 'C', '¥0'};
```

とすることもできます。要するに普通の配列と同じです。しかし、これもちょっと面倒くさいので

```
char str[4] = "ABC";
```

と書くこともできます。もちろん、

```
char str[] = "ABC";
```

と書くこともできます。では、

```
char str[8] = "ABC";
```

はどうでしょうか。配列の初期化のときに

```
int a[4] = {1, 2};
```

とした場合 a[2], a[3] には 0 が入っていましたね。これと同じで str[3] から str[7] までの各要素には '¥0' が入ります。さて、文字配列も普通の配列と同じです。ということは、str[n] は *(str + n) と書いても同じことです。

また、ポインタの場合は

```
char *str;
str = "ABC";
```

とした後で、

```
str = "def";
```

というように値を変更できますね。

配列の場合は

```
char str[] = "ABC";
```

とした後で

```
str = "def";
```

とできるのでしょうか。答えはNoですね。この場合strは配列の名前で、変数ではありません。しかし、各要素の内容は書き換えても問題ありません。

```c
/* strarray01.c */

#include <stdio.h>

int main()
{
    char str[] = "ABC";

    printf("%s¥n", str);

    str[0] = 'd';
    str[1] = 'e';
    str[2] = 'f';
    /* str[3]はもともと'¥0'なので書き換える必要なし */

    printf("%s¥n", str);

    return 0;
}
```

実行結果は次のようになります。文字列の内容が置き換わっていることがわかります。

```
ABC
def
```

8.6 文字列の入力

さて、文字列の扱いに少し慣れてくると、ユーザーに文字列を入力してもらうプログラムを書きたくなりますね。

int型の数値を入力するときは

```
int a;
scanf("%d", &a);
```

のようにscanf関数を使いました。文字列を入力してもらうには

```
char str[16];
scanf("%s", str);
```

のようにします。

「strに&はつけなくていいの？」と言っている人はいませんか？

```
scanf("%d", &a);
```

は、scanf関数に変数aのアドレスを教えていたのでしたね。なぜ教えていたかというと、関数はアドレスを教えてもらわないと、変数の中身を変えることができないからでした。scanf関数はユーザーに値を入力してもらいそれを変数aに代入するので、aのアドレスを知る必要があるのです。

strは配列の名前で先頭要素のアドレスを表していましたね。ということは、わざわざ&をつけなくてもよいのです（&をつけると違う意味になる）。

では、せっかくですからユーザーから文字列を入力してもらい、それを表示するプログラムを作ってみましょう。

```c
/* strinput.c */

#include <stdio.h>

int main()
{
    char szStr[32];

    printf("文字列を入力してください---");
    scanf("%s", szStr);
    printf("あなたの入力した文字列は「%s」ですね¥n", szStr);

    return 0;
}
```

実行結果を見てみましょう。

```
文字列を入力してください---abc
あなたの入力した文字列は「abc」ですね
```

ユーザーがとんでもなく長い(szStr配列に収まりきらない)文字列を入力した場合、結果は予想できません。入力を受け取る配列は、大きめにしておく必要があります。

さて、このプログラムで次のような疑問がわきませんか。

```c
char *str;
scanf("%s", str);
```

としては、だめだろうか?

strはchar型へのポインタですね。ということはアドレスを格納する変数なのです。これはだめですね。

次に、上のstrinput.cプログラムで、「Yasutaka Kumei」のようにスペースの入った文字列を入力するとどうなるでしょうか。scanf関数はスペースを入

力の区切りとみなし、思ったような結果になりません。

そこで登場するのがgets関数です。この関数はスペースも受け取ることができます。改行文字はヌル文字に変換されます。

gets関数

標準入力から文字列を読み込みます。

```
char *gets(
    char *buffer
);
```

インクルードファイルはstdio.hです。

```
/* strinput02.c */

#include <stdio.h>

int main()
{
    char str[64];

    printf("文字列を入力してください--");
    gets(str);
    printf("入力した文字列は「%s」ですね¥n", str);
    return 0;
}
```

実行結果は次のようになります。スペースの入った文字列も大丈夫です。

```
文字列を入力してください--Yasutaka Kumei
入力した文字列は「Yasutaka Kumei」ですね
```

このプログラムを改良して次のようにしてみましょう。文字列を入力すると、「入力した文字列は何々ですね」と表示する。次にまた、文字列を入力するよう要求する。これを繰り返す。やめるときは何も入力せずにEnterキーを押す。

```
/* strinput03.c */

#include <stdio.h>
#include <string.h>

int main()
{
    char str[64];

    while (1) {
        printf("文字列を入力してください---");
        gets(str);
        if (strcmp(str, "") == 0)
            break;
        printf("入力した文字は「%s」ですね¥n", str);
    }

    return 0;
}
```

何も入力せずにEnterキーを押すと、strは""（ヌル文字のみの文字列）となります。このとき、strcmp関数は文字列を比較して同じとみなし、0を返します。break文で無限ループを抜けて、プログラムが終了します。これを、もう少し気の利いたものに書き換えることができます。

```
/* strinput04.c */

#include <stdio.h>
#include <string.h>

```

```
int main()
{
    char str[64], *s ="文字列を入力してください---";

    printf(s);
    while (strcmp(gets(str), "")) {
        printf("入力した文字列は %sですね¥n", str);
        printf(s);
    }

    return 0;
}
```

まず、「printf(s);」という見慣れない書き方が出てきましたね。

```
printf("abc");
```

と書いた場合

```
printf(文字列"abc"の先頭文字のアドレス);
```

の意味でしたね。これを「char *s = "abc";」としたらどうでしょうか。sには"abc" の先頭文字のアドレスが代入されます。したがって「printf(s);」としても同じなのです。

次にwhile文の継続条件式を見てください。これまた見慣れない形ですね。gets関数は戻り値として、引数を返します。引数strには入力した文字列が入っていますね。これは、

```
while(strcmp(str, "")) {
...
}
```

の意味になります。strと "" を比べて同じであればstrcmp関数は0を返します。すると、条件式が偽となりwhileループを抜けるという仕組みです。

8.7 文字列の加工

さて、実用的なプログラムでは文字列の加工が必要となることがよくあります。代表な関数を見てみましょう。

strcpy 関数は文字列をコピーします。

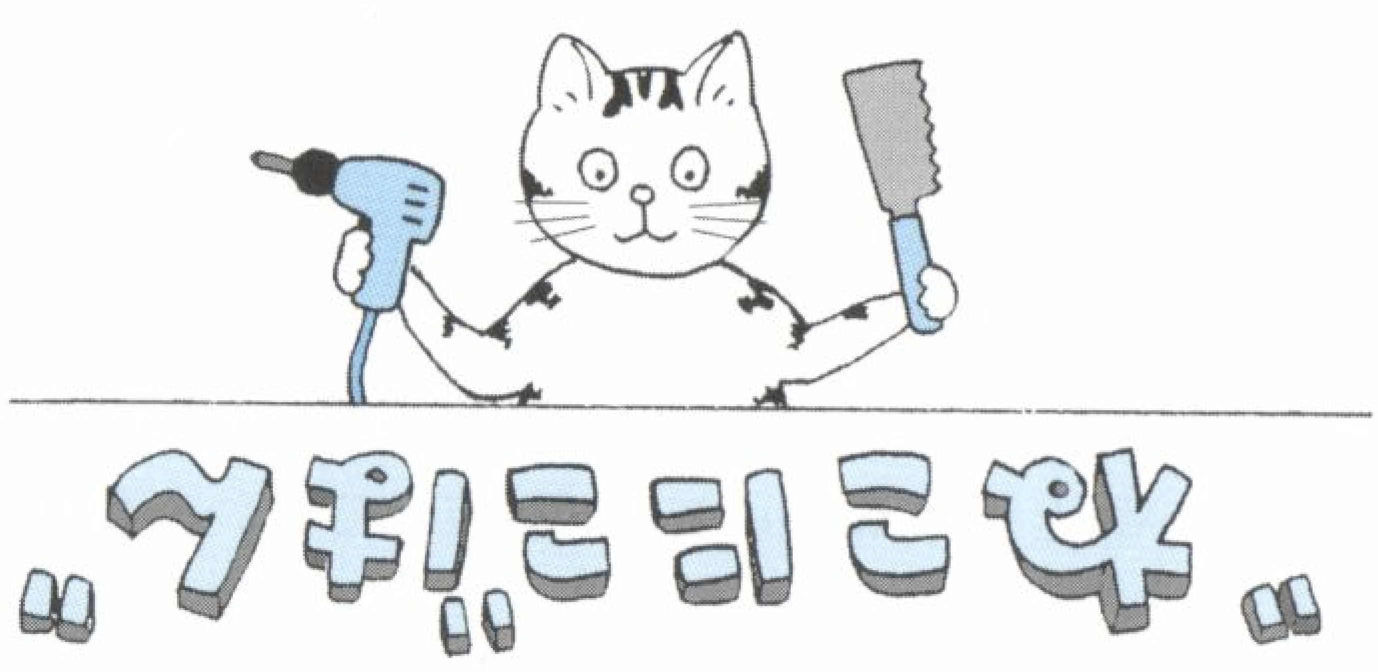
strcpy 関数

文字列をコピーします。

```
char *strcpy(
    char *strDestination,
    const char *strSource
);
```

strDestination に strSource をコピーします。コピーした結果、strDestination から文字列があふれてしまった場合の動作は未定です（要するにまずいことが起こる）。戻り値は、コピー先の文字列（のアドレス）となります。この関数は、文字列の書き換えにも役立ちますね。なお、**const** というのは、その変数の内容を書き換えない、という意味の予約語です。

strarray01.c（194 ページ）のプログラムを書き換えてみましょう。



文字列の加工 … ？

```c
/* strarray02.c */

#include <stdio.h>
#include <string.h>

int main()
{
    char str[] = "ABC";

    printf("%s¥n", str);
    strcpy(str, "def");
    printf("%s¥n", str);

    return 0;
}
```


これだと、長い文字列の書き換えも楽ですね。ただし、strの配列は十分に用意しておく必要があります。

strcat関数は文字列を連結します。

strcat関数

文字列を連結します。

```c
char *strcat(
    char *strDestination,
    const char *strSource
);
```

strDestinationの後ろに、strSourceを連結します。戻り値は連結後のstrDestinationです。サンプルを見てみましょう。

```c
/* strcat01.c */

#include <stdio.h>
#include <string.h>
```

```
int main()
{
    char name[64], sama[] = "様";

    printf("お名前を入力してください。---");
    gets(name);
    printf("%sですね。お待ちしておりました¥n",
        strcat(name, sama));

    return 0;
}
```

実行結果は次のようになります。

```
お名前を入力してください。---田中
田中様ですね。お待ちしておりました
```

もし、このプログラムがわかりにくければ

```
gets(name);
strcat(name, sama);
printf("%sですね。お待ちしておりました¥n", name);
```

と書き換えてみてください。

printfのフォーマットにのっとった加工をする関数もあります。

sprintf関数

書式にのっとって変換した出力を文字列に格納します。

```
int sprintf(
    char *buffer,
    const char *format [,
    argument] ...
);
```

bufferは出力の格納先です。formatは書式指定文字列です(printf関数と同じ)。サンプルを見たほうが早いでしょう。

```
/* sprintf01.c */

#include <stdio.h>
#include <string.h>

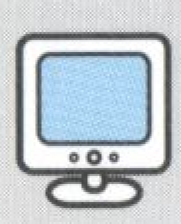
int main()
{
    char buffer[128], name[32];
    int age;
    double bl, bw;

    while (1) {
        printf("氏名---");
        gets(name);
        if (!strcmp(name, "")) {
            printf("氏名が記入されていません¥n");
            continue;
        }
        break;
    }
    while (1) {
        printf("年齢---");
        scanf("%d", &age);
        if (age < 0 || age > 150) {
            printf("年齢が不適切です¥n");
            continue;
        }
        break;
    }
    while (1) {
        printf("身長(cm)---");
        scanf("%lf", &bl);
        if (bl < 0.0 || bl > 250.0) {
            printf("身長の値が不適切です¥n");
            continue;
        }
```

```
        break;
    }
    while (1) {
        printf("体重(kg)---");
        scanf("%lf", &bw);
        if (bw < 0.0 || bw > 200.0) {
            printf("体重の値が不適切です¥n");
            continue;
        }
        break;
    }
    sprintf(buffer,
        "¥n%sさん(年齢は%d歳)のプロフィール¥n"
        "身長は%5.1fcm、体重%5.1fkgで、BMIは%4.1fです¥n",
        name, age, bl, bw, bw * 10000.0 / (bl * bl));
    printf(buffer);

    return 0;
}
```

実行結果を見てみましょう。



```
氏名---糸井康孝
年齢---28
身長(cm)---172
体重(kg)---58

糸井康孝さん(年齢は28歳)のプロフィール
身長は172.0cm、体重 58.0kgで、BMIは19.6です
```

各入力は、while文の無限ループの中で行われています。適切な入力が行われると、break文でループを抜けられます。不適切な入力だと、continue文でループの最初に戻されます。最後に、体裁を整えてprintf関数で文字列を出力しています。

8.8 文字列ポインタを配列にする

次のような配列を考えてみましょう。

```
char *str[6];
```

例によって演算子 * と[ ]の優先順位を考えると、[ ]のほうが順位が上です(62 ページの表 4-1 参照)。したがって

```
char *(str[6]);
```

となります。これは str[6] という char ＊型の配列ですね。すなわち、char 型へのポインタの配列、ということになります(この表現がわかりにくければ、「各要素が char 型へのポインタである配列」)。これを初期化するには、次のように書けばいいですね。

```
char *str[6] = {
    "abc",
    "de",
    "fghi",
    "jklmnop",
    "qrs",
    "tuvwxyz",
};
```

特に、最後の "tuvwxyz" の後にコンマがついてますが、もちろんなくてもかまいません(171 ページ参照)。

```c
/* strings01.c */

#include <stdio.h>

int main()
{
    char *str[6] = {
        "abc",
        "de",
        "fghi",
        "jklmnop",
        "qrs",
        "tuvwxyz",
    };
    int i;

    for (i = 0; i < 6; i++)
        printf(str[i]);

    printf("¥n");

    return 0;
}
```

このプログラムの実行結果を予想してみてください。もうわかりましたね。次のように表示されます。

```
abcdefghijklmnopqrstuvwxyz
```

さて、str[0], str[1], … str[5]は配列なので、隙間なくメモリ上に並んでいるはずです。確かめてみましょう。

```
/* strings02.c */

#include <stdio.h>

int main()
{
    char *str[6] = {
        "abc",
        "de",
        "fghi",
        "jklmnop",
        "qrs",
        "tuvwxyz",
    };
    int i;

    for (i = 0; i < 6; i++)
        printf("str[%d]のアドレス = %p¥n", i, &str[i]);

    return 0;
}
```

実行結果はたまたま次のようになりました。

```
str[0]のアドレス = 0012FEC0
str[1]のアドレス = 0012FEC4
str[2]のアドレス = 0012FEC8
str[3]のアドレス = 0012FECC
str[4]のアドレス = 0012FED0
str[5]のアドレス = 0012FED4
```

きちんと4バイトずつに並んでいますね。

- str[0]には、文字列 "abc" の先頭文字のアドレスが格納されています。
- str[1]には、文字列 "de" の先頭文字のアドレスが格納されています。
- str[2]には、文字列 "fghi" の先頭文字のアドレスが格納されています。

- str[3]には、文字列 "jklmnop" の先頭文字のアドレスが格納されています。
- str[4]には、文字列 "qrs" の先頭文字のアドレスが格納されています。
- str[5]には、文字列 "tuvwxyz" の先頭文字のアドレスが格納されています。

したがって

- str[0][0]には 'a'
- str[0][1]には 'b'
- str[0][2]には 'c'
- str[0][3]には '¥0'

が格納されています。

str[0][4]とか、str[0][5]、str[0][6]、str[0][7]にはアクセスしてはいけませんね。ところが、str[5][4], str[5][5], str[5][6], str[5][7]にはアクセス可能です。

文字列の長さが揃っていないためにこのようなことが起こるのです。プログラムを書くときには、十分注意してください。

8.9 コマンドライン引数

今までは、main関数は引数を取らない関数として扱ってきました。しかし、main関数は次のような引数をとることができるのです。

```
int main(int argc, char *argv[])
```

ところで、main関数の引数はどこからもらってくるのでしょうか。今までの関数は呼び出し元の関数から引数の値をもらってきました。実はmain関数の引数は、コマンドライン引数なのです。ここで、コマンドライン引数についてわからない方もいらっしゃると思いますので、簡単に説明しましょう。

コマンドラインからmain.exeというプログラムを実行するには、コマンドラインに

```
main.exe
```

または、

```
main
```

とタイプして、Enterキーを押しますね。このとき、オプションをつけることができるのです。

```
main -d -x
main readme.txt
main /w
```

などのようにオプションをつけてabcを起動することができます。これは、MS-DOSの時代にはよく行われていました。しかし、Windows時代になっても同じです。プログラム側では、オプションスイッチを読み取って、その内容に合った行動をとります。このスイッチは、決まったものがあるのではなく、プログラムごとに自由に定義します。

コマンドラインから

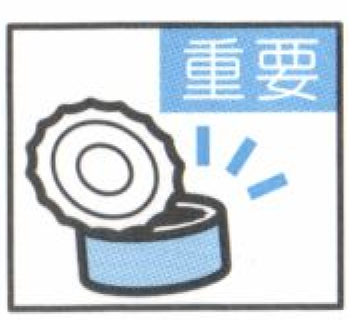
```
main.exe switch01
```

と、タイプしてabc.exeを起動した場合、main関数の引数にはどのような値が入っているのでしょうか。argcには、コマンドラインの引数の数（自分自身の名前も含む）が入っています。この場合2ですね。

また、2番目の引数はどうなっているかというと

argv[0]には、"main.exe"（プログラム自身の名前）
argv[1]には、"switch01"

が格納されます。もうすでにおわかりかと思いますが、argv[0], … argv[n]には、入力した文字列の先頭文字のアドレスが格納されているのです。



argcにはユーザーが入力した文字列の個数、argv[]にはユーザーが入力した文字列が入る。

プログラム側では、これらのオプションスイッチに従って独自の動作をすればよいわけです。では、サンプルを作ってみましょう。

```
/* main01.c */

#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    if (argc == 1) {
        printf("コマンドライン引数がありません¥n");
        return -1;
    }
    for (i = 0; i < argc; i++)
        printf("argv[%d]は「%s」です¥n", i, argv[i]);

    return 0;
}
```

さて、このプログラムにいろいろな引数をつけて起動してみます。実行結果は次のようになりました。後はユーザーが入力したコマンドライン引数に応じた処理を記述すればよいわけです。

```
c:¥sample>main01.exe /d /w readme.txt
argv[0]は「c:¥sample¥main01.exe」です
argv[1]は「/d」です
argv[2]は「/w」です
argv[3]は「readme.txt」です
```

8.10 関数に配列を渡す

関数に配列を引数として渡すにはどうしたらよいのでしょうか。実は前節ですでにやっていますね。前節の main 関数は引数にポインタ配列がありました。もっと簡単な例を見てみましょう。

```
/* param01.c */

#include <stdio.h>

int showarray(int *);

int main()
{
    int a[3] = {10, 20, 30};

    showarray(a);

    return 0;
}

int showarray(int x[])
{
    int i;

    for (i = 0; i < 3; i++)
        printf("x[%d] = %d¥n", i, x[i]);

    return 0;
}
```

プロトタイプ宣言では、

```
int showarray(int *);
```

となっていますが、実際の関数は

```
int showarray(int x[])
```

となっています。要するに、引数がint型へのポインタであるということがわかればよいのです。プロトタイプ宣言に

```
int showarray(int x[]);
```

とか、

```
int showarray(int x[3]);
```

と書いても問題ありません(添え字は無視されます)。関数側についても

```
int showarray(int x[])
int showarray(int x[3])
int showarray(int *x)
```

などの書き方があります。この場合も添え字の３は無視されます。
実行結果は次のようになります。

```
x[0] = 10
x[1] = 20
x[2] = 30
```

もう少し、複雑な例を見てみましょう。

```
/* param02.c */

#include <stdio.h>

int showname(char **);

int main()
{
    char *name[] = {
        "糸井康孝",
        "田中良一",
        "Thomas",
        "エジソン"
    };

    showname(name);

    return 0;
}

int showname(char **p)
{
    int i;

    for (i = 0; i < 4; i++)
        printf("%s¥n", p[i]);

    return 0;
}
```


showname関数の引数はchar型へのポインタのポインタですね。プロトタイプ宣言や関数を書くときは、次のどちらでもよいことになります。

```
int showname(char *p[])
int showname(char **p)
```

では、実行結果を見てみましょう。

```
粂井康孝
田中良一
Thomas
エジソン
```

次に、多次元配列を関数に渡す場合のプログラムを見てみましょう。

```c
/* param03.c */

#include <stdio.h>

int showint(int x[][2]);

int main()
{
    int a[][2] = {1, 2,
        3, 4,
        5, 6,
        7, 8
    };

    showint(a);

    return 0;
}

int showint(int m[][2])
{
    int i, j;

    for (i = 0; i < 4; i++) {
        for (j = 0; j < 2; j++) {
            printf("m[%d][%d] = %d¥n", i, j, m[i][j]);
        }
    }
```

```

    return 0;
}
```

多次元配列の場合、省略できる添え字は最初のもののみでした。引数もこれと同じです。

さて、この例では、showint関数は

```
int showint(int m[][2])  ・・・・・・(1)
```

の他に、

```
int showint(int **) ・・・・・・・・・・(2)
int showint(int (*m)[2])  ・・・・・(3)
```

とも書けます。特に(3)の書き方に注意してください。「*m[2]」と書くと演算子の優先順位の関係より「*(m[2])」の意味になり、意図したものと異なってしまいます。関数に渡したいのはint型へのポインタの配列です。したがって(*m)[2]となるわけです。

では、param03.cの実行結果を見てみましょう。

```
m[0][0] = 1
m[0][1] = 2
m[1][0] = 3
m[1][1] = 4
m[2][0] = 5
m[2][1] = 6
m[3][0] = 7
m[3][1] = 8
```

正しく動作していますね。

練習問題

1. "ABC"のように英字の大文字のみからなる文字列を、"abc"のように小文字に変換するプログラムを作ってください。
2. 1を入力すると「Jan」、2を入力すると「Feb」というように、その月の名前を返すプログラムを文字列の配列を利用して作ってください。
3. 20人のクラスで試験を行いました。試験の受験番号と点数（0～100の整数）を入力して平均値を出すプログラムを作ってください。また、受験番号を入力すると点数が表示されるようにもしてください。受験番号は1番から順に20番までとします。
4. 文字列を反転する（"abcd"を"dcba"にする）関数を作り、これを利用してユーザーの入力した文字列を反転して表示するプログラムを作ってください。ただし、文字列には2バイト文字は含まないものとします。

※解答は付録CD-ROMに収録

第8章のまとめ

1 配列は変数の集まりである。

2 配列名は、配列の先頭要素のアドレスを表す。

3 次元が2以上の多次元配列も作ることができる。

4 文字列も配列として扱うことができる。

5 ポインタの配列を作ることができる。

6 main関数に引数を渡すことができる。

7 配列を引数として関数に渡すことができる。

第9章 構造体

ある人の名前、年齢、身長、体重などの異なったデータ型をひとまとめにして取り扱うことができるとたいへん便利です。これには**構造体**（structure）を使います。

9.1 構造体の宣言

構造体は次のように宣言します。

```
struct 構造体タグ{
    メンバ;
    ...;
};
```

メンバは、この構造体が有しているいろいろな型のデータです。具体例を示すと

```
struct data {
```

```
    char name[32];
    int age;
    double bl;
    double bw;
};
```

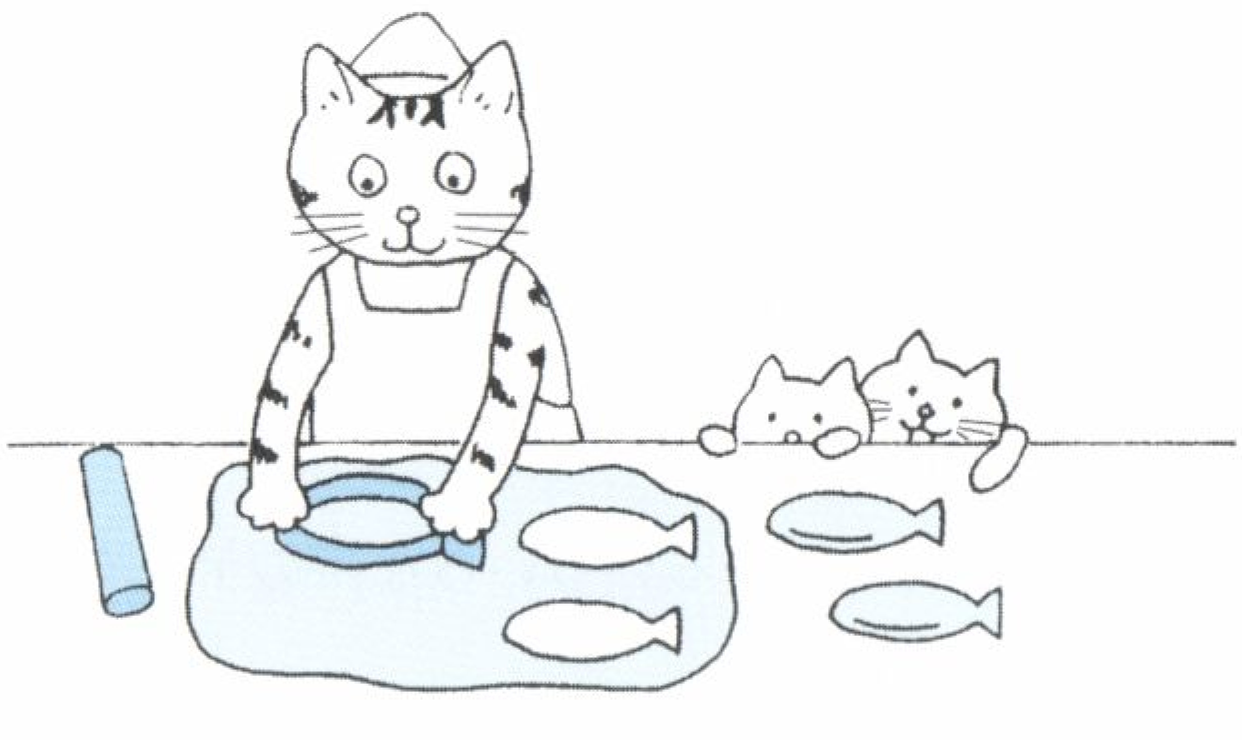
のような感じで宣言します。この構造体にはメンバとしてname，age，bl，bwが存在することがわかります。データ型もさまざまですね。これを構造体の**テンプレート**（template，鋳型）といいます。これは宣言しただけで、まだ使えません。実際に使えるようにするには、構造体タグを使って構造体変数を宣言します。

```
struct data man;
```

のように宣言します。構造体変数manはメンバにname，age，bl，bwを持ちます。テンプレートから複製を作ったわけです。構造体を宣言するということは、見方によっては新しいデータ型を作ったとも考えられます。

また、構造体テンプレートを宣言するのと同時に、構造体変数を宣言することもできます。

```
struct 構造体タグ {
    メンバ;
    ....;
} 構造体変数1，構造体変数2，....;
```



型（テンプレート）からクッキー（変数）を作る

このように、テンプレートと変数を同時に宣言する場合、構造体タグを省略できます。

```
struct {
    メンバ;
    ....;
} 構造体変数1, 構造体変数2,....;
```

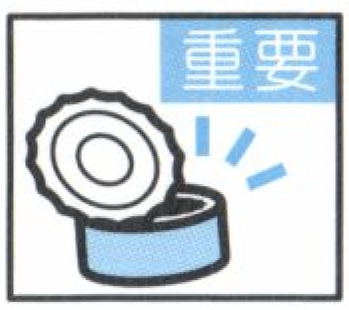
しかし、後から

```
struct 構造体タグ 構造体変数;
```

のように追加で変数を宣言できなくなります。

9.2 構造体メンバへのアクセス方法

さて、構造体変数の宣言の仕方はわかりました。では、具体的にメンバに値を代入したり、メンバの値を読み出したりするにはどうすればよいのでしょうか。これには、**構造体メンバ演算子**(structure member operator)を使います。これはドット(.)です。具体的な使い方をサンプルを見て研究してください。



構造体変数のメンバにアクセスするには、構造体メンバ演算子(.)を使う。

```
/* struct01.c */

#include <stdio.h>
#include <string.h>

struct Man {
```

```
    char name[32];
    int age;
};

int main()
{
    struct Man student;

    strcpy(student.name, "山田太郎");
    student.age = 16;

    printf("生徒の名前は %s です¥n", student.name);
    printf("年は %d 歳です¥n", student.age);

    return 0;
}
```

構造体変数の宣言

構造体メンバ演算子(.)で、メンバにアクセスする

実行結果は次のようになります。

```
生徒の名前は山田太郎です
年は 16 歳です
```

さて、構造体変数は宣言と同時に初期化することもできます。次のサンプルを見てください。

```
/* struct02.c */

#include <stdio.h>

struct DATA {
    int data1;
    char data2;
    char data3[32];
};

int main()
```

```
{
    struct DATA mydata = {10, 'A', "cat"};
    printf("%d, %c, %s¥n",
        mydata.data1, mydata.data2, mydata.data3);
    return 0;
}
```

実行結果は次のようになります。

```
10, A, cat
```

さて、構造体メンバをそっくりそのまま、同じ型の構造体にコピーするにはどうしたらよいのでしょうか。構造体変数a，bに対して単に

```
b = a;
```

とすれば、aのメンバの値はすべてbのメンバにコピーされます。もちろんa，bは同じ型の構造体でなくてはいけません。

```
/* structcopy.c */

#include <stdio.h>

int main()
{
    struct DATA {
        int x;
        int y;
    } a = {10, 20}, b;

    /* 構造体のコピー */
    b = a;

    printf("x = %d, y = %d¥n", b.x, b.y);
```

```

    return 0;
}
```

実行結果は次のようになります。

```
x = 10, y = 20
```

きちんとコピーされていますね。実は昔のC言語ではこのようなことは許されていませんでした。各メンバごとにコピーする必要があったのです。

9.3 構造体の配列

構造体変数の配列を作ることもできます。

```
struct Man student;
```

というように構造体変数を宣言していたところを

```
struct Man student[5];
```

などとすればよいですね。student[2]の name メンバにアクセスするには

```
student[2].name
```

とすればよいわけです。[ ]と . は同じ優先順位の演算子です。こういう場合は結合規則が働きますね。[ ]や . の結合規則は→ですから、「student[2].name」は「(student[2]).name」の意味になります。

では、あるクラスの学生５人の成績を構造体の配列を使って表示するプログラムを作ってみましょう。

```
/* structarray.c */

#define STUDENTNO 5
#define SUBJECTNO 3

#include <stdio.h>

int drawline(void);

struct SEISEKI {
    char name[32];
    int kokugo;
    int sugaku;
    int eigo;
};

int main()
{
    struct SEISEKI myclass[STUDENTNO] = {
        "田中", 80, 80, 55,
        "佐藤", 75, 90, 70,
        "大田", 50, 45, 30,
        "鈴木", 100, 55, 90,
        "加藤", 60, 85, 35
    };
    int i, sum0 = 0, sum1[SUBJECTNO], sum2[STUDENTNO];

    for (i = 0; i < SUBJECTNO; i++)
        sum1[i] = 0;
    for (i = 0; i < STUDENTNO; i++)
        sum2[i] = 0;

    for (i = 0; i < STUDENTNO; i++) {
        sum1[0] += myclass[i].kokugo;
        sum1[1] += myclass[i].sugaku;
        sum1[2] += myclass[i].eigo;
    }

    for (i = 0; i < STUDENTNO; i++)
        sum2[i] = myclass[i].kokugo + myclass[i].sugaku +
```

構造体配列

```
                  myclass[i].eigo;

    for (i = 0; i < STUDENTNO; i++)
        sum0 += sum2[i];


    printf("**** 成績 ****¥n¥n");

    printf("%6s %6s %6s %6s %6s¥n",
        "氏名", "国語", "数学", "英語", "合計");
    drawline();

    for (i = 0; i < STUDENTNO; i++)
        printf("%6s %6d %6d %6d %6d¥n",
            myclass[i].name, myclass[i].kokugo,
            myclass[i].sugaku, myclass[i].eigo, sum2[i]);

    drawline();
    printf("%6s %6.1f %6.1f %6.1f %6.1f¥n",
        "平均",
        (double)sum1[0] / STUDENTNO,
        (double)sum1[1] / STUDENTNO,
        (double)sum1[2] / STUDENTNO,
        (double)sum0 / STUDENTNO);
    return 0;
}

int drawline(void)
{
    int i;

    for (i = 0; i < 35; i++)
        printf("-");
    printf("¥n");

    return 0;
}
```

実行結果は次のようになります。

```
**** 成績 ****

  氏名   国語   数学   英語   合計
--------------------------------
  田中     80     80     55    215
  佐藤     75     90     70    235
  大田     50     45     30    125
  鈴木    100     55     90    245
  加藤     60     85     35    180
--------------------------------
  平均   73.0   71.0   56.0  200.0
```

9.4 構造体へのポインタ

構造体変数を

```
struct Man *student;
```

のように宣言すると、student は Man 構造体へのポインタとなります。このとき、メンバにアクセスするには

```
(*student).name
```

のようにアクセスします。もし、

```
*student.name
```

のように記述するとどうでしょうか。* と . では、. のほうが優先順位が高いため

```
*(student.name)
```

の意味になってしまいます。これは、間違いです。かならず「(*student).name」のようにかっこでくくります。

しかし、**アロー演算子**(->)を使う方が一般的です。

```
student->name
```

と書くことができます。このほうがわかりやすいですね。構造体メンバを関数で書き換えてもらう場合、構造体へのポインタを引数として渡してやる必要がありますね。



構造体ポインタが指す構造体のメンバにアクセスするには、アロー演算子(->)を使う。

```
/* struct03.c */

#include <stdio.h>

struct Data {
    char name[32];
    int age;
    double bl;
    double bw;
    double bmi;
};

int struct_input(struct Data *);

int main()
{
    struct Data MyData;
    char format[] = "%sさん(%d歳)のプロフィール¥n"
        "身長 = %5.1fcm, 体重 = %5.1fkg, BMI = %4.1f¥n";
    char buffer[256];

    struct_input(&MyData);
    sprintf(buffer, format, MyData.name, MyData.age,
        MyData.bl, MyData.bw, MyData.bmi);
    printf(buffer);
```

```
    printf("%s", MyData.name);
    return 0;
}

int struct_input(struct Data *p)
{
    printf("氏名---");
    gets(p->name);
    printf("年齢---");
    scanf("%d", &p->age);
    printf("身長(cm)---");
    scanf("%lf", &p->bl);
    printf("体重(kg)---");
    scanf("%lf", &p->bw);
    p->bmi = p->bw * 10000.0 / (p->bl * p->bl);

    return 0;
}
```

「&p->age」は、演算子&と->では->のほうが優先順位が高いので「&(p->age)」の意味になります。

実行結果を見てみましょう。

```
氏名---Yasutaka Kumei
年齢---30
身長(cm)---178
体重(kg)---75
Yasutaka Kumeiさん(30歳)のプロフィール
身長 = 178.0cm, 体重 =  75.0kg, BMI = 23.7
```

9.5 新しい型を定義する

typedefキーワードを使えば、データ型の名前を自由に定義することができます。

```
typedef 既存の型名 新しい型名;
```

のように使用します。

```
typedef int SEISU;
```

とすれば、intの代わりにSEISUが使えます。

簡単なサンプルプログラムを見てみましょう。

```
/* type01.c */

#include <stdio.h>

typedef int SEISU;

SEISU main()
{
    SEISU n;

    for (n = 0; n < 10; n++)
        printf("%d, ", n);
    printf("¥n");

    return 0;
}
```

SEISUという型名を定義

実行結果は次のようになります。

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

さて、これでは単にintをSEISUに置き換えただけですね。

「だからどうしたっていうの？」

と言われそうですね。実は、構造体にこのtypedefを適用すると、あたかも新しい型ができたかのように見えます。実際、新しい型ができたと考えてもよいでしょう。

```
typedef struct _tagData {
    int age;
    double bl;
    double bw;
} MYDATA, *LPMYDATA;
```

とすると、_tagData型の構造体変数を宣言するとき、

```
MYDATA mydata;
```

と書くことができます。また、この構造体へのポインタを宣言するときも、

```
LPMYDATA lpmydata;
```

と書くことができます（lpdataに * がついていないことに注意してください）。

重要

typedefキーワードを使えば、データ型の名前を定義できる。

では、簡単なサンプルを見てみましょう。

```
/* type02.c */

#include <stdio.h>
```

```c
#include <stdlib.h>
#include <string.h>

typedef struct DATA {
    char name[32];
    char tel[32];
} MYDATA, *LPMYDATA;

int menu();
int meibo_input(LPMYDATA);
int meibo_output(LPMYDATA);

int main()
{
    int selectno, loopend = 0;

    MYDATA mydata = {"", ""};
    LPMYDATA lpMydata = &mydata;

    while (1) {
        selectno = menu();

        switch (selectno) {
            case 1:
                meibo_input(lpMydata);
                break;
            case 2:
                meibo_output(lpMydata);
                break;
            default:
                loopend = 1;
                break;
        }
        if (loopend)
            break;
    }
    return 0;
}
```

typedefで新しい型MYDATA, *LPMYDATAを作る

20行目で構造体変数を宣言すると同時に初期化しています。また21行目で、この構造体変数のアドレスを、この構造体へのポインタlpMydataに代入しています。

その後、whileの無限ループに突入です。

menu関数(自作)の戻り値をselectnoに代入し、switch文で場合分けしています。selectnoが1ならば、meibo_input関数(自作)を呼び出します。2ならば、meibo_output関数(自作)を呼び出します。その他であれば、loopend変数に1を代入してswitch文を抜けます。

loopendが真(0以外)であれば(37行目)、break文が実行されてwhileループを抜けます。もし、「if (loopend)」という書き方がわかりにくければ、「if (loopend != 0)」としても同じことです。

次に、自作関数を順に見ていきましょう。

```
043: int menu()
044: {
045:     char ret[8];
046: 
047:     printf("****** MENU *****¥n");
048:     printf("1:データ入力¥n");
049:     printf("2:データ表示¥n");
050:     printf("0:終了¥n");
051:     printf("*****************¥n");
052:     printf("---->");
053:     gets(ret);
054: 
055:     return atoi(ret);
056: }
```

menu関数の中で使われているatoiは文字列をint型の数値に変換する関数です。たとえば"123" は文字列ですが、これをatoi関数で変換すると数値の123を得ることができます。menu関数では、stringをint型の整数に変換して戻り値としています。「atoi」というのは変な名前の関数ですが「Ascii TO Integer」と覚えておきましょう。

atoi 関数

文字列を整数に変換します。

```
int atoi(
    const char *string
);
```

この関数の仲間としてatof関数もこの際覚えておいてください。

atof 関数

文字列を浮動小数点数に変換します。

```
double atof(
    const char *string
);
```

もし55行目の「return atoi(ret);」がわかりにくければ、変数int no;を用意しておいて、

```
no = atoi(ret);
return no;
```

と書き換えてみてください。

```
058: int meibo_input(LPMYDATA lpData)
059: {
060:     printf("氏名--");
061:     gets(lpData->name);
062:     printf("電話番号--");
063:     gets(lpData->tel);
064:
065:     return 0;
066: }
```

引数のlpDataにはmydata構造体のアドレスが入っています。lpData->nameは、nameメンバですね。

```
gets(lpData->name);
```

は、これでよいのでしょうか。getsの引数にはバッファのアドレスを指定します。nameは配列の名前なので、これでよいのですね。電話番号(lpData->tel)についても同様です。

```
int meibo_output(LPMYDATA lpData)
{
    if (!strcmp(lpData->name, "")) {
        printf("データがありません¥n");
        return -1;
    }
    printf("氏名-- %s¥n", lpData->name);
    printf("電話番号-- %s¥n", lpData->tel);

    return 0;
}
```

構造体からデータを読み出す関数です。特に説明は不要ですね。

では、実行してみましょう。メニューが表示されたら1を選択して、氏名、電話番号を入力します。メニューの2を選択すると、入力された氏名と電話番号が表示されます。再度入力すると、以前のデータは上書きされます。0入力で終了します。実行結果は次のようになります。

```
****** MENU *****
1:データ入力
2:データ表示
0:終了
*****************
---->1
氏名--粂井康孝
```

```
電話番号--012-345-6789
****** MENU *****
1:データ入力
2:データ表示
0:終了
*****************
---->2
氏名-- 粂井康孝
電話番号-- 012-345-6789
****** MENU *****
1:データ入力
2:データ表示
0:終了
*****************
---->0
```

これでは、ちょっとつまらないですね。何人ものデータを入れられるようにするにはどうしたらよいでしょうか。構造体の配列を使えばいいですね。では、このプログラムを改造してみましょう。

```
/* type03.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct DATA {
    char name[32];
    char tel[32];
} MYDATA, *LPMYDATA;

int menu();
int meibo_input(LPMYDATA);
int meibo_output(LPMYDATA);

int main()
{
```

```
018:     int selectno, loopend = 0;
019:
020:     MYDATA mydata[10];
021:     LPMYDATA lpMydata = mydata;
022:
023:     memset(mydata, '¥0', sizeof(MYDATA[10]));
024:
025:     while (1) {
026:         selectno = menu();
027:
028:         switch (selectno) {
029:             case 1:
030:                 meibo_input(lpMydata);
031:                 break;
032:             case 2:
033:                 meibo_output(lpMydata);
034:                 break;
035:             default:
036:                 loopend = 1;
037:                 break;
038:         }
039:         if (loopend)
040:             break;
041:     }
042:     return 0;
043: }
```

20～21行目の

```
MYDATA mydata[10];
LPMYDATA lpMydata = mydata;
```

は何をしているのでしょうか。mydataは配列の名前ですね。配列の名前は先頭要素のアドレスでした。したがって21行目は

```
LPMYDATA lpMydata = &mydata[0];
```

と同じことです。DATA型構造体へのポインタlpMydataに、mydata[0]のアドレスを代入しているわけです。

23行目のmemsetは見慣れない関数ですね。でも、メモリ(mem)に何かをセット(set)する関数ではないかな、と想像できませんか。実はその通りなのです。

memset関数

指定した文字でバッファを埋め尽くします。

```
void *memset(
    void *dest,
    int c,
    size_t count
);
```

この関数は指定したバッファ(dest)を、指定した文字(c)count個で埋め尽くす関数なのです。なお、この関数を使うときはstring.hをインクルードする必要があります。

ここでは、構造体メンバをすべてヌル文字で埋め尽くしています。なぜこのような処理が必要かというと、データ読み出しのとき、データの収められていない構造体配列の要素を指定したときに「データがありません」と表示するときの判断材料にしているからです。

それでは、main関数から呼び出される自作関数を見ていきましょう。

menu関数はtype02.cのmenu関数(231ページ)と同じです。

```
060: int meibo_input(LPMYDATA lpData)
061: {
062:     char strno[8];
063:     int no;
064: 
065:     printf("何番のデータを入力しますか(0-9)--");
066:     gets(strno);
067:     if (strno[0] < '0' || strno[0] > '9') {
068:         printf("入力が不正です¥n");
069:         return -1;
070:     }
071:     strno[1] = '¥0';
```

```
072:     no = atoi(strno);
073:
074:     printf("氏名--");
075:     gets((lpData + no)->name);
076:     printf("電話番号--");
077:     gets((lpData + no)->tel);
078:
079:     return 0;
080: }
```

データ入力のための関数です。

「何番のデータを入力しますか(0−9)−−」と表示した後、ユーザーが入力した文字列をgets関数で取得してstrnoに格納します(66行目)。データの番号とは配列添え字のことです。

次にstrnoの最初の1文字を調べます。これが '0' から '9' の範囲に入っていなければ「入力が不正です」と表示してreturnします。

次に71行目でstrno[1]に '¥0' を代入しています。たとえばユーザーが「112」と入力した場合でも、strno[1]にヌル文字を代入すればこの文字列は "1" と見なされますね。あとは、この文字列を整数値に変換してnoに代入します(72行目)。

さて、引数のlpDataはmain関数のmydata[0]のアドレスです。mydata[n]のアドレスは(mydata + n)ですね。したがってmydata[n]のnameメンバは

```
(mydata + n)->name
```

となります。telメンバについても同様です。

```
082: int meibo_output(LPMYDATA lpData)
083: {
084:     char strno[8];
085:     int no;
086:
087:     printf("何番のデータを読み出しますか(0-9)--");
088:     gets(strno);
```

```
    if (strno[0] < '0' || strno[0] > '9') {
        printf("入力が不正です¥n");
        return -1;
    }
    strno[1] = '¥0';
    no = atoi(strno);

    if (!strcmp((lpData + no)->name, "")) {
        printf("データがありません¥n");
        return -1;
    }
    printf("氏名-- %s¥n", (lpData + no)->name);
    printf("電話番号-- %s¥n", (lpData + no)->tel);

    return 0;
}
```

データの番号を入力するところは、meibo_input関数と同じです。引数のlpDataは、mydata[0]のアドレスなので、後はわかりますね。

では、実行結果を見てみましょう。

```
****** MENU *****
1:データ入力
2:データ表示
0:終了
*****************
---->1
何番のデータを入力しますか(0-9)--0
氏名--粂井康孝
電話番号--012-345-6789
****** MENU *****
1:データ入力
2:データ表示
0:終了
*****************
---->1
何番のデータを入力しますか(0-9)--1
```

```
氏名--田中良一
電話番号--098-756-3210
****** MENU *****
1:データ入力
2:データ表示
0:終了
*****************
---->2
何番のデータを読み出しますか(0-9)--0
氏名-- 条井康孝
電話番号-- 012-345-6789
****** MENU *****
1:データ入力
2:データ表示
0:終了
*****************
---->2
何番のデータを読み出しますか(0-9)--1
氏名-- 田中良一
電話番号-- 098-756-3210
****** MENU *****
1:データ入力
2:データ表示
0:終了
*****************
---->0
```

9.6 メンバに同じ型へのポインタを持つ構造体

「えっ？！ なんですか、この表題は？」

ちょっとわかりにくい表現ですね。構造体メンバには、自分自身と同じ型へのポインタを持たせることができるのです。言葉で説明するより、例を見たほうが早そうですね。

```
struct data {
    char name[32];
    int age;
    struct data *nextdata;
};
```

というような感じです。このような構造体を**自己参照的構造体**(self-referential structure)といいます。テンプレートの作り方はわかったと思いますが、いったいこんなものどうやって使うのでしょうか。

簡単なサンプルを見てみましょう。

```
/* type04.c */

#include <stdio.h>

typedef struct _tagDATA {
    char name[32];
    int age;
    struct _tagDATA *nextdata;
} DATA;

int main()
{
    DATA a = {"粂井", 28, },
         b = {"田中", 35, },
         c = {"鈴木", 18, };
```

自分自身と同じ型(_tagDATA)へのポインタ

```
    DATA *lpdata;

    a.nextdata = &b;
    b.nextdata = &c;
    c.nextdata = NULL;

    for (lpdata = &a; lpdata; lpdata = lpdata->nextdata)
        printf("%s (%d歳)¥n", lpdata->name, lpdata->age);

    return 0;
}
```

aのnextdataメンバにはbのアドレス、bのnextdataメンバにはcのアドレス、そしてcのnextdataメンバにはNULLを代入しておきます。

そうすると、for文でこんなに簡単に、全部の構造体のすべてのメンバを表示することができるのです。

実行結果は次のようになります。

```
粂井 (28歳)
田中 (35歳)
鈴木 (18歳)
```

さて、どうしてこんなことができるのでしょうか。

for文の初期設定式で、lpdataに構造体aのアドレスを代入しています。継続条件式はlpdataです。これがNULLになるとfor文が終了しますね。再設定式では、lpdataにnextdataメンバを代入しています。つまり、次の構造体を指すようになるわけです。

9.7 共用体

1つの記憶領域を確保して、そこにいろいろなデータ型のデータを格納できる機能を持つ**共用体**(union)と呼ばれるデータ型があります。

「なんか、また変なのが出てきたな ……」などと思わないでください。扱い方は構造体とほとんど同じです。

```
union 共用体タグ {
    共用体メンバ;
    ...
} 共用体変数;
```

のような形になります。

しかし、記憶できるのは1つのメンバだけです。あとで別なメンバに値を代入すると、前のメンバの値は無効となります。また、共用体は初期化できません。

では、サンプルを見てみましょう。

```
/* union01.c */

#include <stdio.h>

union myunion {
    int i;
    double d;
    char c;
    char *str;
};

int main()
{
    union myunion u;

    u.i = 200;
    printf("u.i = %d¥n", u.i);

```

```
019:     u.d = 95.25;
020:     printf("u.d = %f¥n", u.d);
021:
022:     u.c = 'A';
023:     printf("u.c = %c¥n", u.c);
024:
025:     u.str = "ABCDE";
026:     printf("u.str = %s¥n", u.str);
027:
028:     return 0;
029: }
```

共用体では、現在どの型が格納されているのかに注意を払わなくてはなりません。また、19行目の「u.d = 95.25;」を実行した後でu.iを表示するようなプログラムを書いてはいけません。では、実行結果を見てみましょう。

```
u.i = 200
u.d = 95.250000
u.c = A
u.str = ABCDE
```


共用体は、単独で使うことはあまりなく、構造体のメンバとして使われるのが普通です。



太郎君の学校の体育大会では、生徒は「100メートル競争」か「パン食い競走」のいずれかに出場しなくてはなりません。生徒名と、出場した競技名、成績をメンバとする構造体を考えてみましょう。100メートル競走ではタイムが、パン食い競走では順位が成績として記録されることにします。

```
/* union02.c */

#define PAN "パン食い競走"
#define KYO "100m競走"
#define NO  6

#include <stdio.h>
#include <stdlib.h>

typedef struct {
    char name[16];
    char kyogi[32];
    union {
        double tm;
        int juni;
    } seiseki;
} DATA, *LPDATA;

int myinput(LPDATA, char *, char *, char *);

int main()
{
    DATA data[NO];
    LPDATA lpdata;
    int i;

    lpdata = data;

    myinput(lpdata + 0, "田中", PAN, "1");
    myinput(lpdata + 1, "佐藤", KYO, "20.5");
    myinput(lpdata + 2, "太田", PAN, "2");
    myinput(lpdata + 3, "加藤", PAN, "4");
    myinput(lpdata + 4, "粂井", KYO, "15.2");
    myinput(lpdata + 5, "鈴木", PAN, "3");
```

```


    printf("%4s %-14s %4s¥n", "氏名", "競技名", "成績");
    printf("---------------------------¥n");
    for (i = 0; i < NO; i++) {
        printf("%4s %-14s ", data[i].name, data[i].kyogi);
        if (strcmp(data[i].kyogi, "100m競走") == 0) {
            printf("%4.1f秒¥n", data[i].seiseki.tm);
        } else {
            printf("%4d位¥n", data[i].seiseki.juni);
        }
    }

    return 0;
}

int myinput(LPDATA p, char *nm, char *kg, char *sk)
{
    strcpy(p->name, nm);
    strcpy(p->kyogi, kg);
    if (!strcmp(kg, PAN))
        p->seiseki.juni = atoi(sk);
    else
        p->seiseki.tm = atof(sk);
    return 0;
}
```

では、実行結果を見てみましょう。

```
氏名 競技名          成績
---------------------------
田中 パン食い競走      1位
佐藤 100m競走       20.5秒
太田 パン食い競走      2位
加藤 パン食い競走      4位
粂井 100m競走       15.2秒
鈴木 パン食い競走      3位
```

練習問題

1. あるクラスで国語と理科の試験を行いました。受験者数は５人です。各受験者の氏名、国語、理科の点数（0 ～ 100 の整数）を格納する構造体配列を作り、氏名と点数を格納してください。また、氏名の一部を入力すると点数を表示するプログラムを作ってください。

2. メンバに身長、体重、BMI（いずれも double 型）を持つ構造体を作り、身長と体重のメンバに適当な値を設定します。そしてこの構造体のポインタを引数として渡したときに、BMI メンバに計算結果を格納する関数を作ってください。BMI は体重（kg）を身長（m）の２乗で割った値です。

※解答は付録 CD-ROM に収録

第９章のまとめ

1 構造体は、異なるデータ型をひとまとめに扱うことができる。

2 構造体のメンバにアクセスするには、構造体メンバ演算子（ . ）を使う。

3 構造体の配列を作ることができる。

4 構造体ポインタからメンバにアクセスするには、アロー演算子（->）を使う。

5 typedef キーワードを使えば、独自のデータ型名を付けられる。

6 構造体メンバには、自分自身と同じ型へのポインタを持たせることができる。

7 共用体の各メンバは同じメモリを共有する。

第10章

ファイル入出力

ファイルの読み書きができると、かなり実用的なプログラミングができます。また、広い意味ではキーボードとか画面、プリンタなどもファイルとして扱います。

10.1 ファイル出力

ファイルを取り扱うには次の手順を踏みます。

1. ファイルのオープン
2. 読み書き
3. ファイルのクローズ

この節では、テキストファイルに書き込みを行う方法を解説します。まず、ファイルのオープンですが、これにはfopen関数を使います。

fopen 関数

ファイルを開きます。

```
FILE *fopen(
    const char *filename,
    const char *mode
);
```

<stdio.h> が必要です。filename には、ファイル名を指定します。mode には、オープンモード（表10-1）を指定します。

表10-1　fopen関数のオープンモード

mode文字列	説明
"r"	読み出し用にオープン。指定したファイルが存在しないとき、関数は失敗する。
"w"	書き込み用にオープン。指定したファイルがすでに存在するときは、これに上書きされる。指定したファイルが存在しない場合は、新規に作られる。
"a"	すでに存在するファイルに追加する。指定したファイルが存在しなければ新規に作成。
"r+"	ファイル読み込み・更新モード。ファイルが存在しないとき、関数は失敗する。
"w+"	読み出し・書き込みモードでオープン。ファイルが存在すると上書きされる。
"a+"	読み出し、追加・更新モード。
"wb"	バイナリで書き込み。
"rb"	バイナリで読み込み。
"ab"	バイナリで追加。

さて、この関数の戻り値の型である「**FILE ***」とは何でしょうか。これは、FILE構造体へのポインタです。FILE構造体はstdio.hの中で定義されています。興味のある人は、stdio.hの中をのぞいてみてください。ここでは、**ファイルポインタ**と呼ぶことにします。

関数が成功するとファイルポインタが返されます。以後このファイルポインタを使って読み書きすることになります。関数が失敗するとNULLが返され

ます。必ず戻り値をチェックしてから、次の作業に移りましょう。

ファイルをクローズするにはfclose関数を使います。

fclose関数

ファイルを閉じます。

```
int fclose(
    FILE *stream
);
```

<stdio.h>が必要です。引数には、openしたときのファイルポインタを指定します。関数が成功したときは0が返されます。関数が失敗したときはEOFが返されます(EOFを−1と定義してあるコンパイラが多い)。

重要

ファイルはfopen関数で開き、fclose関数で閉じる。

さて、オープンしたファイルに書き込むにはどんな関数を使うのでしょうか。ファイル入出力に関係した多くの関数は「f」で始まる名前を持ちます。fはもちろんfileのfです。毎度おなじみのprintf関数にfをつけてfprintf関数とかいうのがあるのでしょうか。実はあります。

fprintf関数

ストリームに整形済みデータを書き込みます。

```
int fprintf(
    FILE *stream,
    const char *format [,
    argument ]...
);
```

<stdio.h> が必要です。最初の引数は、ファイルポインタです。あとは、printf 関数と同じです。

では、test.txt というファイルを作って、その中に「初めてのファイル入出力です」と書き込むプログラムを作ってみましょう。

```
/* file01.c */

#include <stdio.h>

int main()
{
    FILE *fp;

    fp = fopen("test.txt", "w");

    if (fp == NULL) {
        perror("ファイル・オープンに失敗しました¥n");
        return -1;
    } else {
        printf("ファイルを正常にオープンしました¥n");
    }

    fprintf(fp, "初めてのファイル入出力です。¥n");

    /* ファイルのクローズ */
    if (fclose(fp) != 0) {
        perror("ファイルのクローズに失敗しました¥n");
    } else {
        printf("ファイルを正常にクローズしました¥n");
    }

    return 0;
}
```

ファイルポインタ

ファイルを書き込みモードでオープン

実行結果は、次のようになります。

```
ファイルを正常にオープンしました
ファイルを正常にクローズしました
```

test.txtが新規に作られているはずです。これをテキストエディタで開くと…

初めてのファイル入出力です。

と書き込まれているのが確かめられます。

では、もう少し実用的なプログラムを作ってみましょう。ユーザーから氏名、電話番号の入力を受け付け、ファイルに書き込みます。

```
/* file02.c */

#include <stdio.h>
#include <string.h>

int mywrite(FILE *);

int main()
{
    FILE *fp;
    char filename[256];

    printf("ファイル名(拡張子不要)--- ");
    gets(filename);
    strcat(filename, ".txt");

    fp = fopen(filename, "a");  /* 追加モード */
    if (fp == NULL) {
        perror("ファイルのオープンに失敗しました。¥n");
        return -1;
    }
    while (!mywrite(fp)); /* ←この書き方に注意 */

    fclose(fp);
    return 0;
}
```

```
 
int mywrite(FILE *f)
{
    char name[16];
    char tel[16];
    char yesno[8];
    char *format = "%-15s %-15s¥n";

    printf("氏名-- ");
    gets(name);
    printf("電話-- ");
    gets(tel);

    printf(format, name, tel);
    printf("このデータを書き込みますか(y/n):");
    gets(yesno);
    if (strcmp(yesno, "Y") == 0 || strcmp(yesno, "y") == 0)
        fprintf(f, format, name, tel);

    printf("さらにデータを入力しますか(y/n):");
    gets(yesno);
    if (strcmp(yesno, "N") == 0 || strcmp(yesno, "n") == 0)
        return -1;

    return 0;
}
```

では、実行結果を見てみましょう。

```
ファイル名(拡張子不要)--- denwacho
氏名-- 粂井康孝
電話-- 0166-00-0000
粂井康孝        0166-00-0000
このデータを書き込みますか(y/n):y
さらにデータを入力しますか(y/n):y
氏名-- 粂井マイケル
電話-- 281-999-1234
```

```
条井マイケル      281-999-1234
このデータを書き込みますか(y/n):y
さらにデータを入力しますか(y/n):y
氏名-- 田中太郎
電話-- 050-1700-0000
田中太郎          050-1700-0000
このデータを書き込みますか(y/n):y
さらにデータを入力しますか(y/n):n
```

さて、denwacho.txt の中身はどうなっているかというと

```
条井康孝          0166-00-0000
条井マイケル      281-999-1234
田中太郎          050-1700-0000
```

のようになっています。氏名、電話番号が 15 バイト分のスペースに左詰めで記入されています。

さらに、この電話帳に追加することも可能です。再度、file02.exe を起動します。そして、ファイル名に denwacho を指定します。そうすると、今までの denwacho.txt にデータが追加されていきます。



ファイルを使ったら …



必ず閉じること

Column

ストリーム

入出力には「ストリーム」という概念があります。ストリーム（stream）とは、「流れ」という意味です。つまり、情報の流れという意味です。情報の流れは、ファイルからファイルへと流れます。

キーボードや画面も広い意味でのファイルです。

fprintf 関数の第１引数は「FILE *stream」となっていましたね。

この関数を使って、画面に表示させることも可能です。第１引数に標準出力のファイルポインタを指定すればよいのです。

```
/* stdout.c */

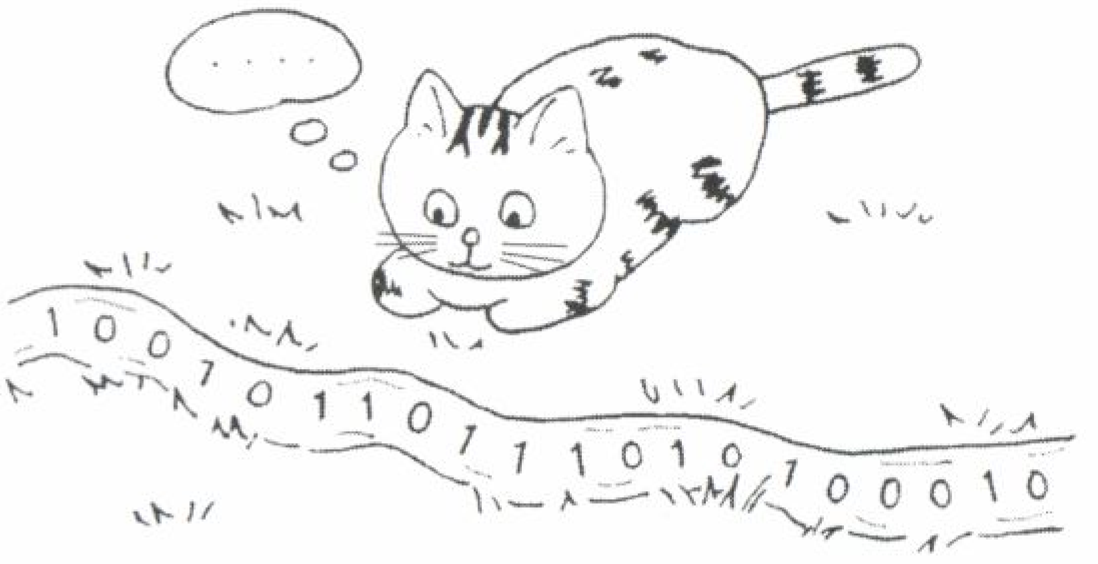
#include <stdio.h>

int main()
{
    fprintf(stdout, "テストです¥n");

    return 0;
}
```

このプログラムを実行してみてください。画面に「テストです」と表示されます。「**stdout**」は標準出力のファイルポインタで、stdio.h の中で定義されています。

常に標準出力に出力する fprintf 版が、printf 関数であるとも考えられます。なお、標準入力は **stdin**，標準エラーは **stderr** と定義されています。



情報の流れ

10.2 ファイル入力

ファイルから読み込む関数には、fscanf関数があります。これもscanf関数の先頭にfがついただけなので、使い方は予想できますね。

fscanf関数

ストリームから整形済みデータを読み込みます。

```
int fscanf(
    FILE *stream,
    const char *format [, argument ]...
);
```

<stdio.h>が必要です。streamにはファイル構造体へのポインタを指定します。formatには変換指定子（93ページの表5-1参照）を指定し、argumentには引数を指定します。この関数は、fileの終端まで来たらEOFを返します。

では、前節で作ったdenwacho.txtを読み取るプログラムを作ってみましょう。

```
/* readfile01.c */

#include <stdio.h>
#include <string.h> /* strcat関数で必要 */

int main()
{
    FILE *fp;
    char filename[256], name[16], tel[16];
    char *format = "%-15s %-15s¥n";

    printf("ファイル名(拡張子不要)--");
    gets(filename);
    strcat(filename, ".txt");
    fp = fopen(filename, "r");
```

```
    if (fp == NULL) {
        perror("ファイルをオープンできません¥n");
        return -1;
    }

    while (fscanf(fp, "%s%s", name, tel) != EOF)
        printf(format, name, tel);

    fclose(fp);
    return 0;
}
```

では、実行してみましょう。

```
ファイル名(拡張子不要)--denwacho
条井康孝        0166-00-0000
条井マイケル    281-999-1234
田中太郎        050-1700-0000
鈴木花子        000-000-1234
```

このプログラムで他のテキストファイルを読み込むとどうなるでしょうか。なんとか読めるが、表示がおかしくなっていますね。denwacho.txtでは、決まった形式でデータが書き込まれています。

readfile01.cでは、ファイルの最初から "%s%s" の変換指定子で読み込んでいます。そして、これらをname, telの配列に読み込んでいますね。

%sで読み込んだ文字列がnameやtelの配列に収まりきらないことも考えられます。

一般的なテキストファイルを読み込むには、1文字読んでは1文字表示する、という方式のほうが安全です。

readfile01.cを書き直してみましょう。

```c
/* readfile01x.c */

#include <stdio.h>
#include <string.h> /* strcat関数で必要 */

int main()
{
    FILE *fp;
    char filename[256];
    int c;

    printf("ファイル名(拡張子不要)--");
    gets(filename);
    strcat(filename, ".txt");
    fp = fopen(filename, "r");
    if (fp == NULL) {
        perror("ファイルをオープンできません¥n");
        return -1;
    }

    while (fscanf(fp, "%c", &c) != EOF)
        printf("%c", c);

    fclose(fp);
    return 0;
}
```

これで、denwacho.txtを読み込んでみましょう。前出のプログラムと同じ結果ですね。他にも、適当なテキストファイルを読み込んでみてください。正しく読み込めますね。

さて、このプログラムをもう少しスマートに書き直してみましょう。

```
001: /* readfile02.c */
002:
003: #include <stdio.h>
004:
005: int main()
006: {
007:     FILE *fp;
008:     char fname[256];
009:
010:     int c;
011:
012:     printf("ファイル名(拡張子不要)---");
013:     fp = fopen(strcat(gets(fname), ".txt"), "r");
014:
015:     if (fp == NULL) {
016:         perror("ファイルがオープンできません¥n");
017:         return -1;
018:     }
019:     while ((c = fgetc(fp)) != EOF)
020:         putc(c, stdout);
021:
022:     return 0;
023: }
```

「あれ？」と思われたところは13行目の「fopen(strcat(gets(fname), ".txt"), "r")」のところではないでしょうか。

gets(fname);でユーザーがファイル名（拡張子なし）を入力します。getsの戻り値はfnameなのです。したがってこれは、fopen(strcat(fname, ".txt"), "r")と同じです。

次に、strcatで拡張子なしのファイル名に「.txt」が付加されました。strcatの戻り値は追加先の文字列です。したがってfopen(fname(.txtが追加されている), "r")と同じなのです。

次にfgetc関数（19行目）という見慣れない関数が出てきました。

fgetc 関数

ファイルから 1 文字ずつ読み込みます。

```
int fgetc(
    FILE *stream
);
```

これは、stream から 1 文字読み込んで int 型で返す関数です。<stdio.h> が必要です。ファイルの終端まで来たら EOF を返します。

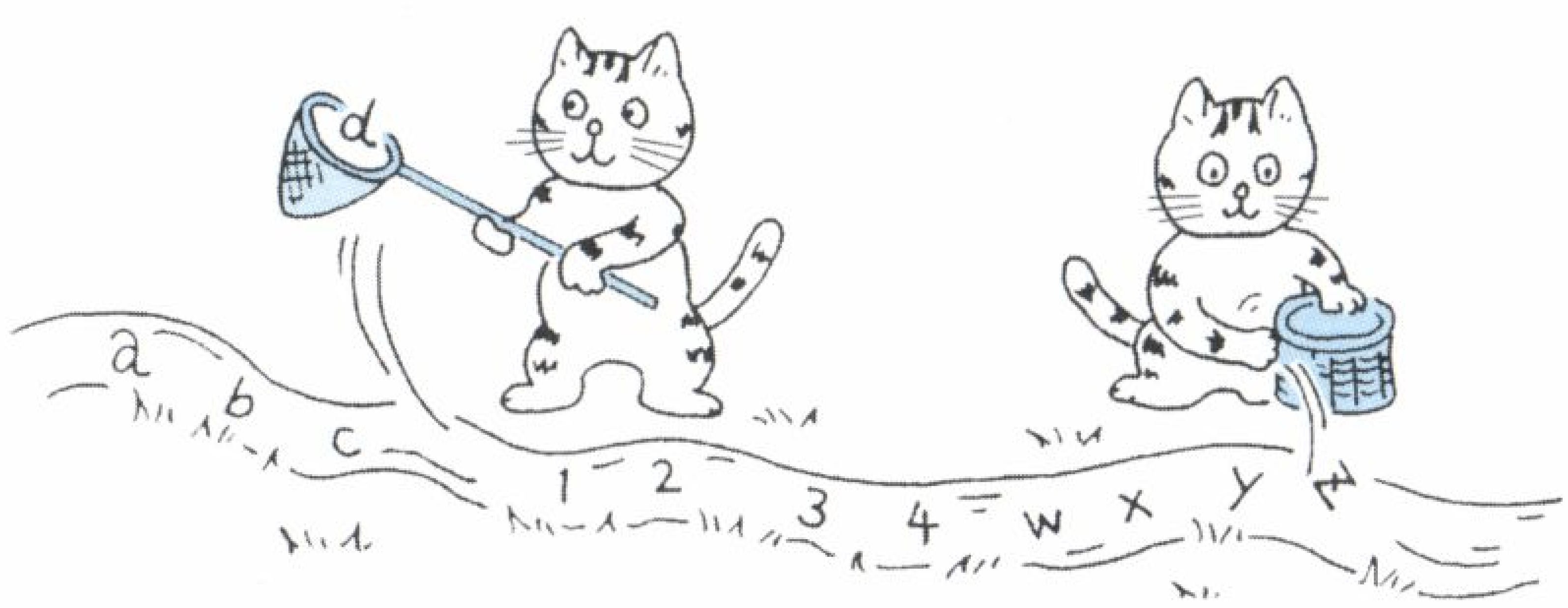
20 行目では putc という新たな関数が使われています。

putc 関数

ファイルに文字を出力します。

```
int putc(
    int c,
    FILE *stream
);
```

putc 関数は、stream に c の 1 文字を出力します。



10.3 ランダムアクセス

ファイルの先頭から順番に読み書きすることを**シーケンシャルアクセス**といいます。ファイルの任意の場所から読み書きできると、より実用的なプログラムを作ることができます。ファイルの任意の位置にアクセスすることを**ランダムアクセス** (random access) といいます。

これは、fseek関数を使えば比較的簡単に実現できます。fで始まる関数なのでファイル関連の関数だなと推理できますね。seekは英語で「探す」という意味です。

fseek関数

ファイルポインタを指定した場所に移動します。

```
int fseek(
    FILE *stream,
    long offset,
    int origin
);
```



シーケンシャル猫は
頭から順に食べる

ランダム猫は
好きなところを食べる

streamにはFILE構造体へのポインタを指定します。offsetには、originからのバイト数を指定します。originには、初期位置を指定します。数値で指定してもよいのですが、次の定数を指定することもできます。

表10-2 originに指定する定数

定数	意味
SEEK_CUR	現在位置
SEEK_END	ファイルの終端
SEEK_SET	ファイルの先頭

しかし、テキストモードで開いたファイルは改行の変換によっておかしな結果となることもあります。テキストモードでも正しく動作するのはoriginがSEEK_SETのときと覚えておきましょう。

それでは、もっとも簡単な例を示します。今まで使ってきたdenwacho.txtをランダムに読み込んでみましょう。その前に、このファイルに書き込まれている形式をもう一度確認してみましょう。

1人分のデータは "%-15s %-15s¥n" の変換指定子で書き込まれています。「名前15バイト＋空白1バイト＋電話番号15バイト＋改行文字(CR-LF)2バイト」です。

つまり、1人分のデータは33バイトですね。n番目の人のデータはファイルの先頭から33×(n − 1)バイト目からということになります(ファイルの先頭は0バイト目)。

```
/* random01.c */

#define RECORDLEN 33

#include <stdio.h>
#include <stdlib.h>

int main()
{
    FILE *fp;
```

```
    char fname[256], data[16], ans[8];
    int no;

    printf("ファイル名(拡張子不要)--");
    fp = fopen(strcat(gets(fname), ".txt"), "r");
    if (fp == NULL) {
        perror("ファイルがオープンできません¥n");
        return -1;
    }
    while (1) {
        printf("何人目のデータを読み込みますか---");
        gets(ans);
        no = atoi(ans);
        fseek(fp, RECORDLEN * (no - 1), SEEK_SET);
        if (fscanf(fp, "%s", data) == EOF) {
            perror("読み込みエラーです¥n");
            continue;
        }
        printf("[氏名] %s", data);
        fscanf(fp, "%s", data);
        printf("[電話] %s¥n", data);
        printf("続けますか(y/n): ");
        gets(ans);
        if (strcmp(ans, "n") == 0 || strcmp(ans, "N") == 0)
            break;
    }

    fclose(fp);

    return 0;
}
```

では、実行結果を見てみましょう。

```
ファイル名(拡張子不要)--denwacho
何人目のデータを読み込みますか---2
[氏名] 条井マイケル[電話] 281-999-1234
続けますか(y/n): y
何人目のデータを読み込みますか---10
読み込みエラーです
: No error
何人目のデータを読み込みますか---3
[氏名] 田中太郎[電話] 050-1700-0000
続けますか(y/n): n
```

fseek関数はファイルの終端を超えた場所を指定してもエラーとはなりません。しかし、fscanf関数は読み込みに失敗してEOFを返します。上の例では3回目にわざと、10人目という存在しないデータを要求してみました。

では、書き込みもランダムにできるプログラムを作ってみましょう。

```
/* random02.c */

#define FNAME "meibo.txt"
#define RECORDLEN 88

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int menu();
int myread();
int mywrite();
int myrewrite();

char *format = "%-15s %-3d %-2s %-63s¥n";

int main()
{
    int menuno;

```

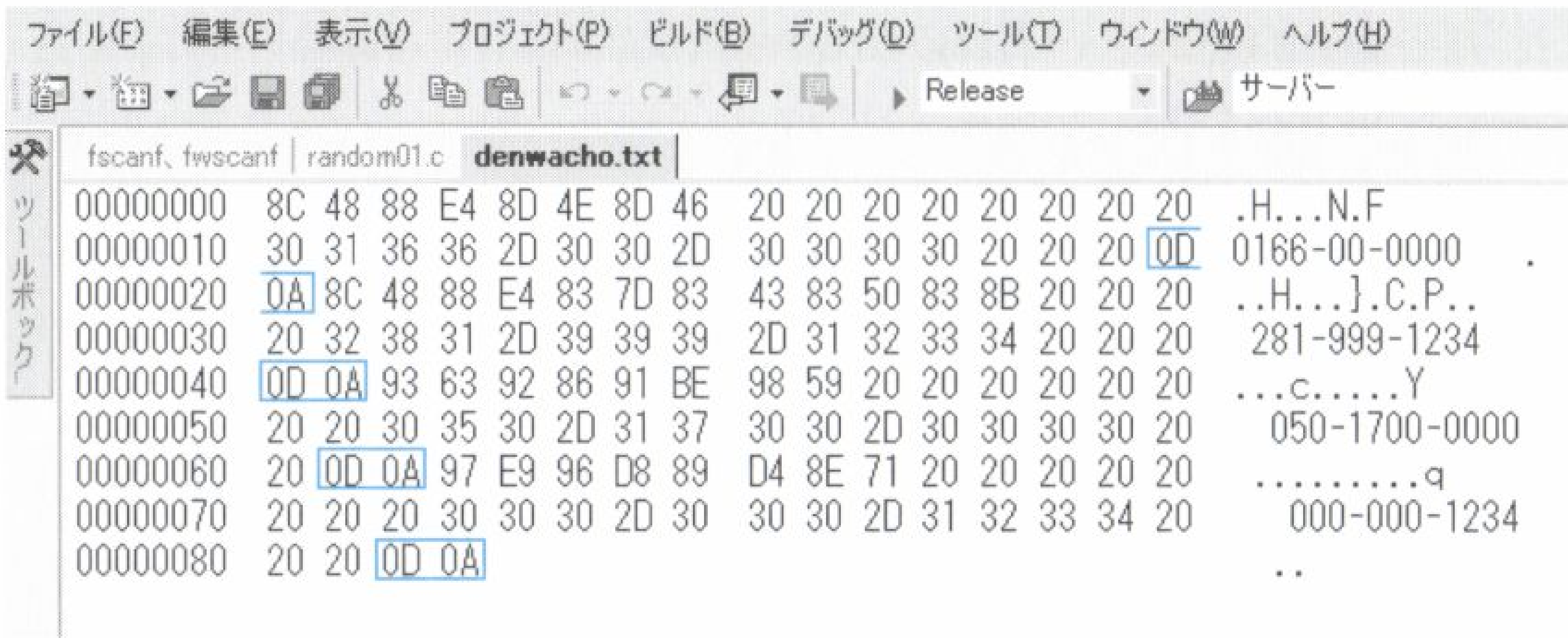
```
021:     while (1) {
022:         menuno = menu();
023:         switch (menuno) {
024:             case 0:
025:                 break;
026:             case 1:
027:                 mywrite();
028:                 break;
029:             case 2:
030:                 myread();
031:                 break;
032:             case 3:
033:                 myrewrite();
034:                 break;
035:             default:
036:                 printf("不正です¥n");
037:                 break;
038:         }
039:         if (!menuno)
040:             break;
041:     }
042: 
043:     return 0;
044: }
```

main関数では、いきなりwhileループに入っています。22行目でmenu関数を呼んで、戻り値をmenunoに代入しています。23行目のswich文ではmenunoの値により場合分けをしています。menunoが0のときは(24行目)何もせずにswitch文を抜けます。39行目でmenunoが0であればbreak文で無限ループを抜けてプログラムを終了します。

Column

改行コード

「fprintf(fp, "...¥n");」のように「¥n」を指定しても、Windowsの場合、改行は「¥r¥n」として扱われます。denwacho.txtを16進ダンプしてみるとよくわかります（先頭が0バイト目）。



上の図はdenwacho.txtをVisual C++ .NETのバイナリエディタで開いたところです。

33×(n－2)バイト目に0D
33×(n－1)バイト目に0A
(n＝0, 1, 2, …)

が来ているのが確かめられます。「¥r」のASCIIコードは13(0x0D)、「¥n」は10(0x0A)です。

しかし、コンパイラによっては改行が「¥n」のみで出力されるものがあるので注意が必要です。

第10章 ファイル入出力

```
int menu()
{
    char ret[8];

    while (1) {
        printf("*******************¥n");
        printf("1:データ書き込み(新規・追加)¥n");
        printf("2:データ読み出し¥n");
        printf("3:データ修正¥n");
        printf("0:終了¥n");
        printf("*******************¥n");
        printf("選択--> ");
        gets(ret);
        ret[1] = '¥0';
        if (ret[0] < '0' || ret[0] > '3') {
            printf("番号が不正です¥n");
            continue;
        }
        return atoi(ret);
    }
}
```

メニューを表示して、ユーザーの選択番号を返します。もし、ユーザーが「28」と入力したならば、これは「2」とみなされます。

```
int mywrite()
{
    FILE *fp;
    char name[16], sex[4], address[64], buffer[8];
    int age;

    fp = fopen(FNAME, "a");
    if (fp == NULL) {
        perror("ファイルをオープンできません¥n");
        return -1;
    }
    while (1) {
```

```
080:         printf("氏名--");
081:         gets(name);
082:         printf("年齢--");
083:         gets(buffer);
084:         age = atoi(buffer);
085:         printf("性別(M/F)--");
086:         gets(sex);
087:         printf("住所--");
088:         gets(address);
089: 
090:         if (fprintf(fp, format, name, age, sex, address) < 0) {
091:             perror("書き込みエラーが発生しました");
092:             break;
093:         }
094:         printf("さらに入力を続けますか(y/n): ");
095:         gets(buffer);
096:         if (buffer[0] == 'n' || buffer[0] == 'N')
097:             break;
098:     }
099: 
100:     fclose(fp);
101:     return 0;
102: }
```


ファイルに書き込む関数です。74行目で、ファイルを追加モードでオープンしています。すでにファイルが存在していれば、これに追加します。80～93行目で、氏名、年齢、性別、住所を尋ねて、これをファイルに書き込んでいます。

```
104: int myread()
105: {
106:     FILE *fp;
107:     char search[16], yesno[8], name[16], sex[4], address[64];
108:     int no = 0, find = 0, age;
109: 
110:     fp = fopen(FNAME, "r");
111:     if (fp == NULL) {
```

```
112:         perror("ファイルをオープンできません¥n");
113:         return -1;
114:     }
115: 
116:     printf("検索する氏名-- ");
117:     gets(search);
118: 
119:     while (1) {
120:         fseek(fp, RECORDLEN * no++, SEEK_SET);
121:         if (fscanf(fp, "%s", name) == EOF)
122:             break;
123:         if (strstr(name, search) != NULL) {
124:             find++;
125:             fscanf(fp, "%d", &age);
126:             fscanf(fp, "%s", sex);
127:             fscanf(fp, "%s", address);
128:             printf("氏名: %s¥n", name);
129:             printf("年齢: %d¥n", age);
130:             printf("性別: %s¥n", sex);
131:             printf("住所: %s¥n", address);
132:             printf("------------------¥n");
133:             printf("さらに検索を続けますか(Y/N):");
134:             gets(yesno);
135:             if (yesno[0] == 'y' || yesno[0] == 'Y')
136:                 continue;
137:             else
138:                 break;
139:         }
140:     }
141:     printf("%d件が検索されました¥n", find);
142:     fclose(fp);
143:     return 0;
144: }
```

ファイルから読み込む関数です。氏名の一部を入力すると、それを探し出して表示します。たとえば「田」で探すと「太田」「田中」などが引っかかってきます。

110行目でファイルを読み込み専用モードでオープンしています。119行目から無限ループに入っています。120行目のfseek関数の2番目の引数に注

意してください。

```
RECORDLEN * no++
```

* と ++ では ++ のほうが優先順位が高いのです。一番最初に fseek が呼ばれるときは、no はまだ 0 のままです。ループを回って次に fseek が呼ばれると no は 1 となります。このループを抜けるのは fscanf 関数が EOF を返したとき（ファイルの終端に達したとき）（121 行目）と、ユーザーが「検索を続けますか」の質問に対して続けないと答えたときです。

まず、氏名をファイルから name に読み込みます。これと、ユーザーが入力した検索文字列（search）を比較します（123 行目）。

文字列の検索には、strstr 関数を利用しています。

strstr 関数

文字列を検索します。

```
char *strstr(
    const char *string,
    const char *strCharSet
);
```

<string.h> が必要です。string に検索対象文字列を指定します。strCharSet に検索したい文字列を指定します。string の中に strCharSet が見つかった場合は、そのポインタが返されます。複数ある場合は、最初に見つけた場所となります。文字列が見つからなかった場合は、NULL が返されます。

もし、ユーザーの指定した文字列が見つかった場合は、まず find を 1 増やします（124 行目）。次に、年齢、性別、住所を読み込みます（125 ～ 127 行目）。そして、氏名、年齢、性別、住所を表示して（128 ～ 131 行目）、検索を続けるかどうか尋ねます。検索を続けないのであれば、ループを抜けます。

```
146: int myrewrite()
147: {
148:     FILE *fp;
```

```c
    char buffer[8], shusei[16], yesno[8],
         name[16], sex[4], address[64];
    int no = 0, age, find = 0;

    fp = fopen(FNAME, "r+");
    if (fp == NULL) {
        perror("ファイルをオープンできません¥n");
        return -1;
    }
    printf("修正するデータの氏名-- ");
    gets(shusei);
    while (1) {
        fseek(fp, RECORDLEN * no++, SEEK_SET);
        if (fscanf(fp, "%s", name) == EOF)
            break;
        if (strstr(name, shusei) != NULL) {
            find = 1;
            fscanf(fp, "%d", &age);
            fscanf(fp, "%s", sex);
            fscanf(fp, "%s", address);
            printf("氏名: %s¥n", name);
            printf("年齢: %d¥n", age);
            printf("性別: %s¥n", sex);
            printf("住所: %s¥n", address);
            printf("------------------¥n");
            printf("このデータを修正しますか(Y/N):");
            gets(yesno);
            if (yesno[0] == 'y' || yesno[0] == 'Y') {
                printf("氏名を修正しますか(Y/N):");
                gets(yesno);
                if (yesno[0] == 'y' || yesno[0] == 'Y') {
                    printf("氏名--");
                    gets(name);
                }
                printf("年齢を修正しますか(Y/N):");
                gets(yesno);
                if (yesno[0] == 'y' || yesno[0] == 'Y') {
                    printf("年齢--");
                    gets(buffer);
                    age = atoi(buffer);
```

```
189:                 }
190:                 printf("性別を修正しますか(Y/N):");
191:                 gets(yesno);
192:                 if (yesno[0] == 'y' || yesno[0] == 'Y') {
193:                     printf("性別--");
194:                     gets(sex);
195:                 }
196:                 printf("住所を修正しますか(Y/N):");
197:                 gets(yesno);
198:                 if (yesno[0] == 'y' || yesno[0] == 'Y') {
199:                     printf("住所--");
200:                     gets(address);
201:                 }
202:                 fseek(fp, RECORDLEN * (no - 1), SEEK_SET);
203:                 fprintf(fp, format, name, age, sex, address);
204:             }
205:         }
206:     }
207:     if (find == 0)
208:         printf("修正すべきデータはありませんでした¥n");
209:
210:     fclose(fp);
211:     return 0;
212: }
```

ファイル中のデータを修正する関数です。

153行目でファイルを "r+" モードでオープンしている点に注意してください。

ユーザーが修正するデータの氏名の一部を入力すると、この文字列を含むデータを検索します(161行目)。これは、myread関数と同じ要領です。

修正すべきデータが見つかったら(164行目～)、氏名や住所などの項目を修正するかどうか尋ねます。そして、新しいデータを入力してもらいます。

最後に、fseekで書き込む場所を指定して(202行目)、書き込みます。ここで注意すべき点は、修正すべき箇所はnoを1引いてやらないと、次のデータに書き込みがされてしまうということです。

では、実行結果を見てみましょう。まずは、ファイルがない状態でデータ入力を行います。

```
********************
1:データ書き込み(新規・追加)
2:データ読み出し
3:データ修正
0:終了
********************
選択--> 1
氏名--条井志麻
年齢--22
性別(M/F)--F
住所--旭川市
さらに入力を続けますか(y/n): y
氏名--太田良子
年齢--25
性別(M/F)--F
住所--東京都
さらに入力を続けますか(y/n): y
氏名--条井康孝
年齢--28
性別(M/F)--M
住所--東京都
さらに入力を続けますか(y/n): y
氏名--マイケル
年齢--10
性別(M/F)--M
住所--家の中
さらに入力を続けますか(y/n): n
********************
1:データ書き込み(新規・追加)
2:データ読み出し
3:データ修正
0:終了
********************
選択--> 0
```

この時点での meibo.txt は、次のようになっています。

```
条井志麻      22  F  旭川市
太田良子      25  F  東京都
条井康孝      28  M  東京都
マイケル      10  M  家の中
```

では、再度プログラムを起動して検索を行ってみましょう。

```
********************
1:データ書き込み(新規・追加)
2:データ読み出し
3:データ修正
0:終了
********************
選択--> 2
検索する氏名-- 条井
氏名: 条井志麻
年齢: 22
性別: F
住所: 旭川市
------------------
さらに検索を続けますか(Y/N):y
氏名: 条井康孝
年齢: 28
性別: M
住所: 東京都
------------------
さらに検索を続けますか(Y/N):y
2件が検索されました
********************
1:データ書き込み(新規・追加)
2:データ読み出し
3:データ修正
0:終了
********************
選択--> 0
```

確かに正しく検索されていますね。さて、ここで「粂井康孝」の住所が間違っていることに気がつきました。

```
********************
1:データ書き込み(新規・追加)
2:データ読み出し
3:データ修正
0:終了
********************
選択--> 3
修正するデータの氏名-- 康孝
氏名: 粂井康孝
年齢: 28
性別: M
住所: 東京都
------------------
このデータを修正しますか(Y/N):y
氏名を修正しますか(Y/N):n
年齢を修正しますか(Y/N):n
性別を修正しますか(Y/N):n
住所を修正しますか(Y/N):y
住所--北海道
********************
1:データ書き込み(新規・追加)
2:データ読み出し
3:データ修正
0:終了
********************
選択--> 0
```

これで、「粂井康孝」の住所が修正されているはずです。meibo.txtの中身をのぞくと次のようになっていました。

```
粂井志麻    22 F 旭川市
太田良子    25 F 東京都
粂井康孝    28 M 北海道
マイケル    10 M 家の中
```

確かに修正されていますね。

10.4 バイナリファイル

今まで取り扱ってきたファイルはテキストファイルでした。バイナリファイルとは、コンピュータ内部で使用されるデータ形式をそのまま書き込んであるファイルです。テキストエディタなどで開いても、意味のある文章にはなりません。テキストファイルでは、123という数値も"123"という文字列として書き込まれています。

バイナリファイルの入出力は、基本的にはテキストファイルのそれと同じ方針でよいのですが、ファイルオープンモードや読み込み·書き込みのときに使用する関数が異なります。

1. fopen関数でファイルをオープンする。オープンモードは"rb", "wb", "ab"などbの文字が入ったものとなります。
2. 書き込みにはfwrite関数、読み込みにはfread関数を使います。
3. 読み書きが終了したらfclose関数でファイルをクローズします。

fwrite関数

ストリームにデータを書き込みます。

```
size_t fwrite(
    const void *buffer,
    size_t size,
    size_t count,
    FILE *stream
);
```

bufferには、書き込むデータへのポインタを指定します。sizeには、項目のサイズをバイト単位で指定します。普通はsizeof演算子を利用するのでsize_t型となっています。countには、書き込む最大項目数を指定します。文

字列などを書き込む場合はstrlen関数を利用するので、size_t型となっています。streamには、FILE構造体へのポインタを指定します。戻り値は、実際に書き込んだ項目数となります。

fread関数

ストリームからデータを読み込みます。

```
size_t fread(
    void *buffer,
    size_t size,
    size_t count,
    FILE *stream
);
```

bufferには、データの格納場所を指定します。sizeには、項目のバイト数を指定します。countには、読み出す最大の項目数を指定します。streamには、FILE構造体へのポインタを指定します。戻り値は、実際に読み出した項目数となります。



- バイナリファイルの書き込みにはfwrite関数を使う。
- バイナリファイルの読み込みにはfread関数を使う。

では、試しにバイナリモードでファイルに書き込んでみましょう。

```
/* binary01.c */

#include <stdio.h>

int main()
{
    FILE *fp;
    int a[3] = {1, 2, 3};

```

```
    fp = fopen("mybinary.b", "wb");
    if (fp == NULL) {
        perror("ファイルがオープンできません¥n");
        return -1;
    }
    fwrite(a, sizeof(int), 3, fp);
    fclose(fp);
    printf("ファイルに書き込みました¥n");

    return 0;
}
```

これを実行するとmybinary.bというファイルが作られます。この中身をテキストエディタで開いてみてください。何だかわけがわかりませんね。では、これをバイナリエディタで開くとどうなるでしょう。図10-1を見てください。

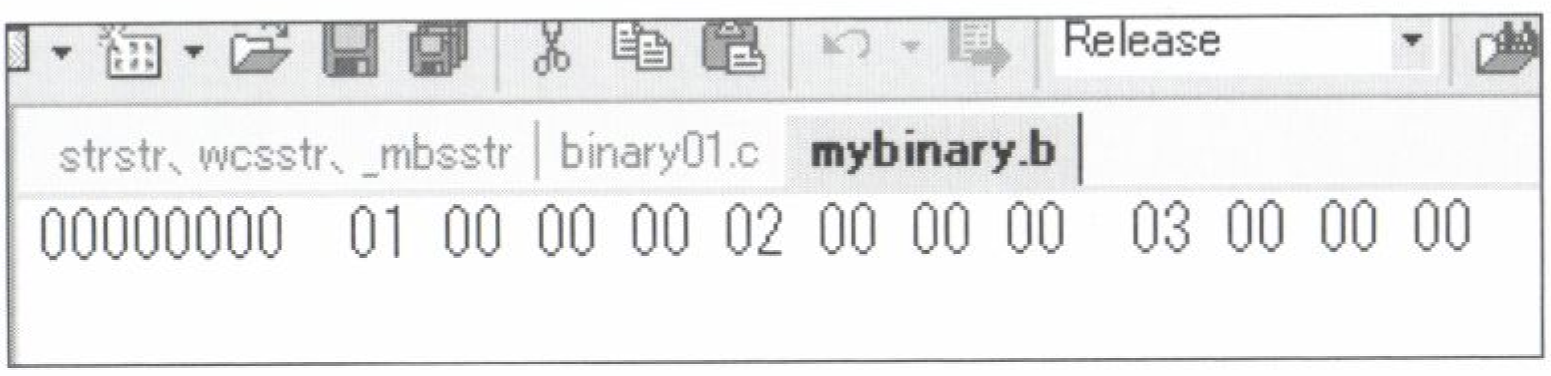


図10-1 **バイナリエディタ**

なんとなくわかりましたね。int型は4バイトです。4バイトを1かたまりとみると、

```
01 00 00 00
02 00 00 00
03 00 00 00
```

となっていますね。しかも筆者のコンピュータのCPUはIntel製です。つまり、バイトオーダーはリトルエンディアンが採用されています。この並びをビッグエンディアンで書き直すと、

```
00 00 00 01
00 00 00 02
00 00 00 03
```

となり、わかりやすくなります。「大人(BIG)のほうが素直」と覚えるのでしたね(16ページ)。

では、文字列なども書き込んでみましょう。

```
/* binary02.c */

#include <stdio.h>

int main()
{
    FILE *fp;
    char *str = "ABC";
    int a[3] = {14, 15, 16};

    fp = fopen("binary.b", "wb");
    fwrite(str, sizeof(char), strlen(str), fp);
    fwrite(a, sizeof(int), 3, fp);
    fclose(fp);

    printf("書き込み終了¥n");
    return 0;
}
```

これを実行するとbinary.bというファイルが作られます。これをバイナリエディタで見てみると、次のようになっています。

```
41 42 43 0E 00 00 00 0F 00 00 00 10 00 00 00
```

最初に書き込んだのは、1バイト文字3つです。次がint型の数字3つですね。もうわかりましたね。最初の3バイトは、それぞれA, B, CのASCIIコードです。次は4バイトずつ見ていってビッグエンディアン方式で書き直すとわかりやすくなります。

```
41              (A)
42              (B)
43              (C)
00 00 00 0E     (14)
00 00 00 0F     (15)
00 00 00 10     (16)
```

さて、この理屈がわかると、読み出し方法もわかりますね。このbinary.bというバイナリファイルを読み出すプログラムを作ってみましょう。

```
/* binary03.c */

#include <stdio.h>

int main()
{
    FILE *fp;
    int a[3];
    char buf[8];

    memset(buf, '¥0', sizeof(buf[8]));

    fp = fopen("binary.b", "rb");
    fread(buf, sizeof(char), 3, fp);
    fread(a, sizeof(int), 3, fp);
    fclose(fp);
    printf("%s, %d, %d, %d¥n", buf, a[0], a[1], a[2]);

    return 0;
}
```

実行結果は次のようになります。

```
ABC, 14, 15, 16
```

では、この際バイナリエディタもどきを作ってしまいましょう。これは、読み込みの機能しかありません。ですから「エディタ」というのはおこがましいので「もどき」をつけています。

```c
/* bedit.c */

#include <stdio.h>

int main()
{
    FILE *fp;
    char fname[256];
    unsigned char c;
    int no = 0;

    printf("ファイル名---");
    gets(fname);

    fp = fopen(fname, "rb");
    if (fp == NULL) {
        perror("ファイルオープンに失敗¥n");
        return -1;
    }
    while(fread(&c, sizeof(char), 1, fp) != 0) {
        printf("%02X ", c);
        no++;
        if (no % 8 == 0)
            printf("¥n");
    }
    printf("¥n");
    fclose(fp);
    return 0;
}
```

内容は説明するまでもありませんね。では、これで先ほどのbinary.bを開いてみましょう。

```
ファイル名---binary.b
41 42 43 0E 00 00 00 0F
00 00 00 10 00 00 00
```

ファイル入出力ができるようになると、かなり実用的なプログラムが作れます。プログラムにエラーが発生した場合にログファイルに記録することも可能でしょう。家計簿ソフトのようなものも作ることができますし、住所録だって作れますね。

練習問題

1. 氏名、住所、電話番号をテキストファイルに書き込むプログラムを作ってください。各項目はコンマ(,)で区切って書き込みます。また、氏名の一部を入力すると、このファイルから住所と電話番号を読み取って表示する機能も付け加えてください。

※解答は付録CD-ROMに収録

第10章のまとめ

1 入出力関数を使うには、<stdio.h> をインクルードする。

2 ファイルを開くには fopen 関数を使う。

3 ファイルを閉じるには fclose 関数を使う。

4 ファイルにデータを書き込むには fprintf 関数を使い、読み込むには fscanf 関数を使う。

5 ファイルの先頭から順番に読み書きすることをシーケンシャルアクセスと呼ぶ。

6 ファイルの任意の位置にアクセスすることをランダムアクセスと呼ぶ。

7 バイナリファイルを開くときはオープンモードを変更する。

8 バイナリファイルにデータを書き込むには fwrite 関数を使い、読み込むには fread 関数を使う。

第11章

高度なプログラミング

11.1 ポインタを返す関数

関数は戻り値としてポインタを返すこともできます。たとえばchar型へのポインタを返す関数funcは

```
char *func(引数,...);
```

のように宣言します。実用的なプログラムではポインタを返す関数がよく使われます。

では、strcat関数のように、文字列を結合して、戻り値として結合された文字列のポインタを返す関数を作ってみましょう。

```
/* mystrcat01.c */

#include <stdio.h>

char *mystrcat(char *, char *);
```

```
int main()
{
    char str1[16] = "abc";
    char *str2 = "def";

    printf("%s¥n", mystrcat(str1, str2));

    return 0;
}

char *mystrcat(char *a, char *b)
{
    char *ptr;
    int n = 0;

    ptr = a + strlen(a);

    while(*(ptr + n) = *(b + n) ) {
        n++;
    }

    return a;
}
```

実行結果を見てみましょう。

```
abcdef
```

strcat関数と同様、文字列を追加できましたね。

mystrcat関数では何をしているのでしょうか。ポインタptrはaよりstrlen(a)だけ進んだアドレスを指しています。つまり、文字列aの最後の文字（ヌル文字）を指しています。while文の継続条件式が偽となるまでn++が実行されます。これが偽になるのは*(ptr + n)に '¥0' が代入されたときですね。

継続条件式が偽になるまでnは1ずつ増えていきます。つまり、文字列aの後ろに文字列bを先頭から1文字ずつコピーしています。これが終了するのは、文字列bのヌル文字がコピーされたときです。

さて、mystrcat01.cはもっと簡潔に書くことができます。

```
/* mystrcat02.c */

#include <stdio.h>

char *mystrcat(char *, char *);

int main()
{
    char str1[32] = "Hello, ";
    char str2[16] = "World!¥n";

    printf(mystrcat(str1, str2));

    return 0;
}

char *mystrcat(char *a, char *b)
{
    char *aorg;

    aorg = a;

    while (*a)
        a++;
    while (*a++ = *b++);

    return aorg;
}
```

実行結果は次のようになります。

```
Hello, World!
```

なぜこのようなことができるのかわからない人は、もう一度ポインタのところを読み直してみてください。

11.2 関数へのポインタ

変数と同様に関数も実行時にはメモリ上のどこかに置かれているはずです。実は、関数の名前は、関数のアドレスを表しています。

```
/* funcptr01.c */

#include <stdio.h>

int func(int);

int main()
{
    printf("main関数のアドレス = %p¥n", main);
    printf("func関数のアドレス = %p¥n", func);

    return 0;
}

int func(int x)
{
    printf("x = %d¥n", x);

    return 0;
}
```

これを実行すると、main関数やfunc関数のアドレスがわかります。実行結果は次のようになりました。

```
main関数のアドレス = 00401020
func関数のアドレス = 00401000
```

では、関数へのポインタはどのように宣言したらよいのでしょうか。

```
戻り値の型 (*ptr)();
```

のように宣言します。引数は関係ありません。int型が戻り値である関数へのポインタは次のように宣言します。

```
int (*ptr)();
```

では、確認してみましょう。

```
/* funcptr02.c */

#include <stdio.h>

int func(int);

int main()
{
    int (*ptr1)(), (*ptr2)();

    ptr1 = main;
    ptr2 = func;

    printf("main関数のアドレス = %p, ptr1 = %p¥n", main, ptr1);
    printf("func関数のアドレス = %p, ptr2 = %p¥n", func, ptr2);

    return 0;
}

```

```
int func(int x)
{
    printf("x = %d¥n", x);
    return 0;
}
```

実行結果は次のようになりました。

```
main関数のアドレス = 004113B1, ptr1 = 004113B1
func関数のアドレス = 004114B5, ptr2 = 004114B5
```

もちろんポインタで与えられた関数を実行することもできます。

```
int (*ptr)();
```

このポインタが指している関数を実行するには、たとえば次のようにします。

```
(*ptr)(a, b);
```

「実際問題、こんな変なことをする場面はあるのか??」
という疑問がわきませんか。実はちゃんとあるのです。**DLL** (Dynamic Link Library) というのをご存じでしょうか。この中には、関数が機械語に翻訳されて、すぐに実行できる形で納められています。よく使う関数などは、このDLLに納めておけばプログラムのサイズを小さくすることができます。

DLL中の関数をプログラム中で呼び出すには、「暗黙の呼び出し」と「明示的呼び出し」というのがあります。明示的呼び出しでは、プログラム側でその関数が必要になったとき初めて呼び出します。このとき、その関数のアドレスを取得するというわけです。

今までは、関数のプロトタイプ宣言をプログラムの最初のほうの、関数外に書きました。こうすることによって、すべての関数からその関数が見えるからです。

これを、関数の中で宣言することも可能です。ポインタで与えられた関数の実行を確認するのと同時に、関数内でのプロトタイプ宣言の例を見てみましょう。

```
/* funcptr03.c */

#include <stdio.h>

int main()
{
    int myfunc(char *, char *);
    int (*ptr)();

    ptr = myfunc;

    (*myfunc)("田中", "学生");
    (*ptr)("山田", "会社員");
    myfunc("佐藤", "公務員");

    return 0;
}

int myfunc(char *str1, char *str2)
{
    printf("%sさんは、%sです¥n", str1, str2);
    return 0;
}
```

実行結果は次のようになります。

```
田中さんは、学生です
山田さんは、会社員です
佐藤さんは、公務員です
```

さて、このプログラムでは、main関数より手前にある関数からはmyfunc関数が見えません。したがって次のプログラムはエラーとなります。

```
#include <stdio.h>

int yourfunc()
{
    myfunc("後藤", "浪人");    /* コンパイラには
                                  myfunc関数がわからない */
    return 0;
}

int main()
{
    int myfunc(char *, char *);
    ......
    return 0;
}

int myfunc(char *str1, char *str2)
{
    .....
    return 0;
}
```

さて、当然のことながら関数へのポインタを配列にすることも可能です。

「あーーー、次から次にいろいろ変なことを思いつくもんだ。」

別に筆者が思いついたわけではありません。関数へのポインタ配列は次のように宣言します。

```
関数の戻り値の型 (*ptr[要素の個数])();
```

では、サンプルを見てみましょう。

```
/* funcptr04.c */

#define PAI 3.141592653589793

#include <stdio.h>
#include <stdlib.h>

double sankaku();
double daikei();
double en();

int main()
{
    double (*fn[3])() = {sankaku, daikei, en};
    char ret[8];
    int fnno;

    while (1) {
        printf("どの面積を求めますか¥n"
                "(1:三角形 2:台形 3:円 0:終了)--");
        gets(ret);
        if (ret[0] < '0' || ret[0] > '3') {
            printf("番号が不正です¥n");
            continue;
        }
        ret[1] = '¥0';
        fnno = atoi(ret);
        if (fnno == 0)
            break;
        printf("面積は %f です¥n", (*fn[fnno - 1])());
    }
    return 0;
}
```

関数へのポインタ配列は、宣言すると同時に初期化しています(14行目)。

ユーザーに0～3を入力させますが、ユーザーがこれらの数字以外の入力を行うかもしれません。ユーザーの入力をいったん文字列として受け取ります(21行目)。受け取った文字列の先頭の1字が '0'～'3' の範囲になければ、即、入力ミスとわかります。そこで、その旨表示してcontinueします(23～24行目)。

先頭文字が '0'～'3' の範囲にあれば、ret[1]にヌル文字を代入します(26行目)。これで、たとえばユーザーが "19a" などと入力しても "1" になります。

atoi関数を使って、これを整数に変換します(27行目)。変換後、これが0ならば、break文でループを抜けてプログラムを終了します(28～29行目)。後は、(*fn[fnno - 1])()を実行して、その戻り値を表示します。ユーザーの選んだ番号により、実行される関数が異なるというわけです。

sankaku関数は三角形の面積を求めるものです。

```
double sankaku()
{
    char str[32];
    double teihen, takasa;

    printf("三角形の面積を求めます¥n");
    printf("底辺 = ");
    gets(str);
    teihen = atof(str);
    printf("高さ = ");
    gets(str);
    takasa = atof(str);

    return teihen * takasa / 2.0;
}
```

daikei関数は台形の面積を求めるものです。

```
double daikei()
{
    char str[32];
    double jotei, katei, takasa;

    printf("台形の面積を求めます¥n");
    printf("上底 = ");
    gets(str);
    jotei = atof(str);
    printf("下底 = ");
    gets(str);
    katei = atof(str);
    printf("高さ = ");
    gets(str);
    takasa = atof(str);

    return (jotei + katei) * takasa / 2.0;
}
```

en関数は円の面積を求めるものです。

```
double en()
{
    char str[32];
    double hankei;

    printf("円の面積を求めます¥n");
    printf("半径 = ");
    gets(str);
    hankei = atof(str);

    return (hankei * hankei) * PAI;
}
```

このプログラムを実際に動かして、動作を確認してみてください。

11.3 可変個の引数を持つ関数

今まで作った関数は初めから引数の数が決まっていました。引数の数を状況に応じて変えることはできるのでしょうか。実は可能です（不可能だったらこの節ないからね）。

手順としては次のようになります。

1. stdarg.h をインクルードしておきます。
2. 関数の形は次のようになります。「...」は実際にピリオド３つです（オプション引数）。
   関数の戻り値の型 関数名(必須引数, ...);
3. 関数内で va_list 型変数（ポインタ）を宣言します。
4. 「va_start(3.で宣言したポインタ, 必須引数);」と「va_end(3.で宣言したポインタ);」という２つのマクロで挟まれた部分で、オプション引数を取得します。
5. 最初のオプション引数は「va_arg(3.で宣言したポインタ, オプション引数の型);」というマクロで取得します。va_arg マクロをもう一度呼ぶと次のオプション引数が得られます。va_arg マクロの呼び出しが、実際のオプション引数の数より多い場合は何が起こるかわかりません。

では、実際にプログラムを作ってみましょう。ここでは、引数に int 型のオプション引数を渡して、その合計を返す関数を作ってみます。オプション引数の数は、必須引数で指定するものとします。

```
/* variable01.c */

#include <stdio.h>
#include <stdarg.h>

int myfunc(int a, ...);

int main()
```

```
{
    int wa;

    wa = myfunc(4, 1, 2, 3, 4);
    printf("wa = %d¥n", wa);
    wa = myfunc(2, 100, -45);
    printf("wa = %d¥n", wa);
    wa = myfunc(6, 10, 20, 30, 40, 20, 33);
    printf("wa = %d¥n", wa);

    return 0;
}

int myfunc(int x, ...)
{
    va_list ptr;
    int i, sum = 0;

    va_start(ptr, x);
    for (i = 0; i < x; i++)
        sum += va_arg(ptr, int);

    va_end(ptr);

    return sum;
}
```

実行結果は次のようになります。

```
wa = 10
wa = 55
wa = 153
```

どうやら、正しく機能しているようですね。しかし、この関数はいちいちオプション引数の個数を指定しなくてはならず、不便ですね。そこで、「引数の終わりはこれだ！」という合図を決めておくのも１つの方法です。

次の mystrcat 関数は

```
mystrcat("abc", "def", "ghijk", "");
```

のように指定すれば、文字列をどんどん連結していく関数です。引数の最後はヌル文字のみの文字列という約束にします。では、実際にプログラムを作ってみましょう。

```
/* strcat2.c */

#include <stdio.h>
#include <stdarg.h>
#include <string.h>

char *strcat2(char *, ...);

int main()
{
    char str1[256] = "今日は、";  /* 十分な領域を確保しておく */
    char *str2 = "よい天気ですが、";
    char *str3 = "明日はどうなるかわかりません¥n";
    strcat2(str1, str2, str3, "");
    printf(str1);

    return 0;
}

char *strcat2(char *p, ...)
{
    va_list ptr;
    char *st;

    /* 必須引数がヌル文字の文字列であればすぐに戻る */
    if (p[0] == '¥0')
        return p;

    /* ヌル文字の文字列を見つけたらループ脱出 */
    va_start(ptr, p);
    while (1) {
```

```
        st = va_arg(ptr, char *);
        if (st[0] == '¥0')
            break;
        strcat(p, st);
    }
    va_end(ptr);

    return p;
}
```

では、実行結果を見てみましょう。

```
今日は、よい天気ですが、明日はどうなるかわかりません
```

Column

printf 関数の構文

本書をここまで読み進んだ方は、printf 関数の使い方はもう十分に理解できていることでしょう。しかし構文はまだ紹介していませんでした。ここで改めて構文を見てみましょう。

printf 関数

標準出力に整形済みデータを出力します。

```
int printf(
   const char *format [,
   argument]...
);
```

<stdio.h> が必要です。format には書式指定文字列を指定します。ご存知のように、ここに書式指定フィールドを含めることができます。この書式指定フィールドの個数と同じだけ、argument に引数を指定します。引数は省略されることもあれば、1 つのこともあり、複数個の場合もあります。つまり printf 関数は可変個の引数を持つ関数なのです！

11.4 メモリの動的確保

配列の大きさは、

```
int a[10];
```

のように宣言することにより、決まりました。これは、あらかじめint型の変数が10個必要である、とわかっていることが前提となります。しかし、実際のプログラミングでは、この個数がわからないことがよくあります。

ユーザーに整数型データを入力してもらい、その平均値を出すプログラムを考えてみましょう。入力したデータをint型の配列にため込むとして、いくつの要素を用意すればよいのでしょうか。ユーザーは何個のデータを入力するかはわからないですね。

余裕を持って

```
int a[1000];
```

と宣言しますか？ ユーザーがデータを3つしか入力しなかったら、メモリの無駄遣いです。もしかしてユーザーが1000個より多いデータを入力したらどうなるでしょうか。

C言語では、必要なときに変数や配列などのメモリを確保する機能があります。これを**メモリの動的確保**と呼びます。これを実現するにはmalloc関数やcalloc関数を使います。

malloc関数

メモリを取得します。

```
void *malloc(
    size_t size
);
```

malloc関数を使うには、malloc.hをインクルードします。「void *」型とは、どんな型にでもキャストできるという意味です。sizeには、割り当てるバイト

数を指定します。数値でxxバイトというように割り当ててもよいのですが、通常はsizeof演算子を利用するので、sizeの型はsize_t型となっています。たとえばint型1個分の領域を確保するには、次のようにします。

```
int *pi;
pi = (int *)malloc(sizeof(int));
```

また、この関数が失敗したらNULLが返されます。プログラマは、malloc関数の戻り値がNULLでないことを確認する必要があります。

さて、このように動的に確保したメモリは、不要になったらプログラマの責任で解放しなくてはいけません。これにはfree関数を使います。

free関数

メモリを解放します。

```
void free(
    void *memblock
);
```

free関数を使うには、malloc.hをインクルードする必要があります。memblockには、解放するメモリブロックを指定します。



必要なときに必要なだけ確保

重要

動的に確保したメモリは、不要になったらfree関数で解放する。

では、もっとも簡単なサンプルを見てみましょう。

```
001: /* malloc01.c */
002:
003: #include <stdio.h>
004: #include <malloc.h>
005:
006: int main()
007: {
008:     int *pi;
009:
010:     /* int型1個分のメモリブロックを確保 */
011:     pi = (int *)malloc(sizeof(int));
012:
013:     if (pi == NULL) {
014:         perror("メモリの確保に失敗しました。¥n");
015:         return -1;
016:     }
017:
018:     *pi = 100;
019:
020:     printf("*pi = %d¥n", *pi);
021:
022:     /* 不要になったメモリは解放する */
023:     free(pi);
024:
025:     return 0;
026: }
```

14行目では、まだ見たことのない関数が使われています。

perror関数

エラーメッセージを出力します。

```
void perror(
    const char *string
);
```

perror関数は、標準エラー(stderr)に文字列を出力します。この関数を使うには、‹stdio.h›か‹stdlib.h›をインクルードする必要があります。標準エラーとは通常は画面のことです。printfは、標準出力に文字列を出力しました。通常、標準出力は画面です(ここではコマンドプロンプトウィンドウ)。しかし、リダイレクトすれば、標準出力が画面とは限りません。そこで、重要なエラーなど、どうしても画面に出力したい場合はperror関数を使います。

```
perror("abc");
```

が実行されるとabcの後にコロン(:)、スペース、エラーメッセージ(処理系によりいろいろ)、そして改行が出力されます。

malloc01.c実行結果は次のようになります。

```
*pi = 100
```

もう少し実用的なプログラムを作ってみましょう。

```
/* malloc02.c */

#include <stdio.h>
#include <malloc.h>

int main()
{
    int *pi, no, i;
```

```c

    printf("何個の整数値を入力しますか--");
    scanf("%d", &no);
    pi = (int *)malloc(sizeof(int) * no);
    if (pi == NULL) {
        perror("メモリ確保に失敗しました¥n");
        return -1;
    }
    for(i = 0; i < no; i++) {
        printf("%d個目のデータ---", i + 1);
        scanf("%d", pi + i);
    }
    printf("入力されたデータは次の通りです¥n");
    for (i = 0; i < no; i++)
        printf("No.%d = %d¥n", i + 1, *(pi + i));

    free(pi);

    return 0;
}
```

このプログラムを実行してみましょう。

```
何個の整数値を入力しますか--5
1個目のデータ---1234
2個目のデータ---5000
3個目のデータ---2564
4個目のデータ---100
5個目のデータ---6987
入力されたデータは次の通りです
No.1 = 1234
No.2 = 5000
No.3 = 2564
No.4 = 100
No.5 = 6987
```

malloc 関数に似た関数に calloc 関数というのがあります。

calloc 関数

メモリ配列を確保し、0で初期化します。

```
void *calloc(
    size_t num,
    size_t size
);
```

num 個の要素を持つ配列を確保します。各要素のサイズは size バイトです。また、各要素は 0 で初期化されます。不要になったら free 関数で解放するのは malloc 関数と同じです。

```
/* calloc01.c */

#include <stdio.h>
#include <malloc.h>
#include <string.h>

int main()
{
    char *pc;

    pc = (char *)calloc(16, sizeof(char));
    if (pc == NULL) {
        perror("メモリの確保に失敗しました¥n");
        return -1;
    }

    strcpy(pc, "あいうえお");
    printf("%s¥n", pc);

    free(pc);

    return 0;
}
```

メモリを確保し
0で初期化

実行結果は次のようになります。

```
あいうえお
```

Column リダイレクト

次のプログラムは何をしているのでしょうか。

```
#include <stdio.h>

int main()
{
    printf("abc¥n");
    return 0;
}
```

「画面に abc と表示するプログラムに決まっている！」
では、このプログラムが abc.exe であれば、コマンドラインから

```
abc.exe > abc.txt
```

と入力してみましょう。すると、なんと abc.txt というファイルが作られて、この中に abc と書き込まれます。「>」はコマンドライン上で「リダイレクト演算子」と呼ばれる演算子なのです（これは OS の機能です）。

11.5 確保領域の大きさを変更する

さて、動的にメモリを確保した後、その大きさを変更したい場合があります。この場合realloc関数を使います。

realloc 関数

メモリの再確保を行います。

```
void *realloc(
    void *memblock,
    size_t size
);
```

<stdlib.h>か<malloc.h>をインクルードする必要があります。

memblockは、すでに割り当てられているメモリブロックへのポインタです。sizeには、新しいサイズをバイト単位で指定します。

では、サンプルを見てみましょう。

```
/* realloc01.c */

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <stdlib.h>

int main()
{
    double *pd, *pd2, sum = 0;
    int no = 0, i;
    char szData[32];

    pd = (double *)malloc(0);
    if (pd == NULL) {
        perror("領域確保に失敗¥n");
```

```
        exit(-1);
    }

    while (1) {
        printf("数値を入力してください。(Eで終了):");
        gets(szData);
        if (strcmp(szData, "E") == 0 ||
            strcmp(szData, "e") == 0) {
            printf("入力を終了します¥n");
            break;
        }

        pd2 = (double *)realloc(pd, sizeof(double) * (no + 1));
        if (pd2 == NULL) {
            perror("領域のサイズ変更に失敗しました¥n");
            free(pd); /* すでに確保してあるメモリを解放 */
            exit(-2);
        }
        pd = pd2;
        pd[no] = atof(szData);
        sum += pd[no];
        no++;
    }
    if (no) {
        for (i = 0; i < no; i++)
            printf("[No.%d] %f¥n", i + 1, pd[i]);
        printf("合計¥t%f¥n", sum);
        printf("平均¥t%f¥n", sum / no);
    }

    free(pd);
    return 0;
}
```

realloc 関数で、確保領域の大きさを変更

最初に malloc 関数で 0 バイトの領域を確保しておきます（14 行目）。失敗したら標準エラーにその旨表示して exit しています（「malloc(0)」が NULL を返す処理系では「malloc(1)」としてください）。

exit 関数

プログラムを正常に終了させます。

```
void exit(
    int status
);
```

exit 関数は、プログラムを正常に終了させます。プログラム終了時に status がシステムに返されます。このとき、開いているファイルがあれば自動的に閉じられます。確保している領域も自動的に解放されます。

次に、while ループの中身をのぞいてみましょう。21 行目で「数値を入力してください」と表示した後、22 行目の gets 関数でユーザーの入力した文字列を取得しています。文字列が "E" または "e" であれば、break 文で while ループを抜けます。それ以外の場合は数値入力と考えて、realloc 関数で確保領域のサイズを変更します。realloc 関数が失敗したら exit するので、

```
pd = (double *)realloc(...);
```

としてもよいのですが、ここではあえて別のポインタを使用しています。というのも、万一 realloc 関数が失敗したら pd に NULL が入ってしまいます。そうすると、すでに確保してある領域に対して free 関数が無効になってしまうからです。

realloc 関数が成功したら「pd = pd2;」としています。そして、確保した領域に入力された文字列を数値に変換して代入します（36 行目）。37 行目の「sum += pd[no];」で合計値も計算しています。これらの作業が終わったら no を 1 増やしてループの最初に戻ります。47 行目でメモリを解放してプログラムを終了します。では、実行結果を見てみましょう。

```
数値を入力してください。(Eで終了):12.5
数値を入力してください。(Eで終了):13.4
数値を入力してください。(Eで終了):25.26
数値を入力してください。(Eで終了):0.6
数値を入力してください。(Eで終了):-258.2
数値を入力してください。(Eで終了):125
数値を入力してください。(Eで終了):23.56
数値を入力してください。(Eで終了):e
入力を終了します
[No.1] 12.500000
[No.2] 13.400000
[No.3] 25.260000
[No.4] 0.600000
[No.5] -258.200000
[No.6] 125.000000
[No.7] 23.560000
合計     -57.880000
平均     -8.268571
```

このプログラムで、ユーザーが数値以外の文字列を打ち込んだらどうなるでしょうか。実験してみてください。

11.6 分割コンパイル

プログラムを作るとき、ソースファイルは複数あっても問題ありません。機能ごとにソースファイルを分けてプログラミングするということも可能です。a1.c, a2.c, a3.cという3つのソースからひとつのプログラムを作るにはどうすればよいでしょう。

Visual C++の場合は、プロジェクトにこの3つのファイルを参加させてやるだけです。

このとき、問題となるのが変数のスコープです。

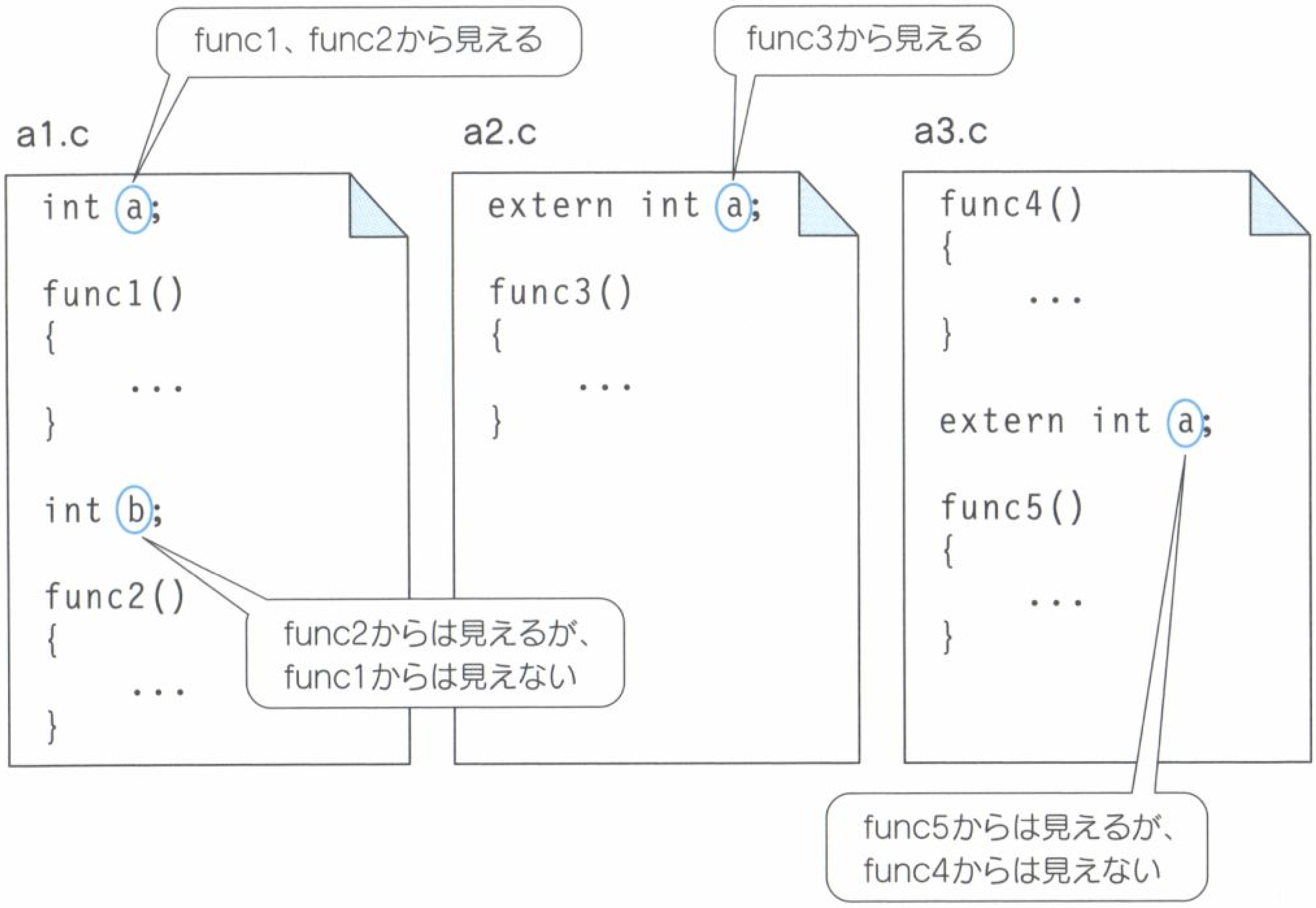


図 11-1　複数のソースファイル

図 11-1 を見てください。変数 a, b は a1.c のグローバル変数です。a1.c の中では、この宣言以降であればどこからでも見えます。しかし、func1 からは b が見えません (func1 より後で b が宣言されているため)。func2 からは、a も b も見えます。

a2.c の最初で「extern int a;」と宣言してあるので、このファイルのすべての関数から a は見えます。

a3.c では、func4 からは a が見えません。func5 からは a が見えます。

なんとなくわかったでしょうか。

では、実際に複数ソースファイルのプログラムを作ってみましょう。

```
/* source1.c */

#include <stdio.h>
```

```
int func1();
int func2();

int a = 100;

int main()
{
    extern char *str;

    printf("ここはmain関数です¥na = %d¥n", a);
    printf("*strも見えます---%s¥n", str);
    func1();
    func2();

    return 0;
}
```

このファイルには、func1，func2 の定義がありません。また、str もないですね。でも、main 関数内で extern 宣言してあるので、どこかのファイルに char str が宣言してあるのでしょう。

```
/* source2.c */
#include <stdio.h>

extern int a;

int func1()
{
    extern char *str;

    printf("ここは、source2.cのfunc1関数¥na = %d¥n", a);
    printf("strも見えます---%s¥n", str);
    return 0;
}
```

printf関数を使うのでstdio.hのインクルードは必要です。

このファイルでは、aは最初に書いてあるので、このファイルにあるすべての関数から参照できます。strは、func1内でextern宣言してあるので、この関数内のみで参照可能です。

```
/* source3.c */

#include <stdio.h>

int func2()
{
    extern int a;
    extern char *str;  /* ·····(1) */

    printf("ここは、source3.cのfunc2関数です¥na = %d¥n", a);
    printf("%s¥n", str);      /* (1)がないとエラーとなる */
    return 0;
}

char *str = "abc";
```

あれ、変なところに「char *str = "abc";」がありますね。ここで宣言しているため、通常ではfunc2からは見えません。ところが、func2内でextern宣言しているので、func2からもstrは見えます。

実行結果を見てみましょう。

```
ここはmain関数です
a = 100
*strも見えます---abc
ここは、source2.cのfunc1関数
a = 100
strも見えます---abc
ここは、source3.cのfunc2関数です
a = 100
abc
```

どうです。分割コンパイルも難しくはないですね。

11.7 プリプロセッサ

#includeとか#defineなどはすでに何回もプログラムに登場していますが、これらはコンパイラが本格的に翻訳を始める前に処理されます。この処理を行うプログラムを**プリプロセッサ**(preprocessor)と呼びます。

```
#include <ファイル名>
#include "ファイル名"
```

これらは、ファイル名で指定されたファイルをそっくりそのままソースファイルに取り込みます。<>を使うと、コンパイラが指定した場所にファイルを探しに行きます。""を使うと、通常、ソースファイルのあるディレクトリを探し(コンパイラにより異なる)、なければコンパイラの指定した場所を探します。

自分で作ったヘッダファイルには""を使い、stdio.hのような既存のヘッダファイルには<>を使うと覚えておきましょう。

```
#define マクロ名 文字列
```

は、ソースファイル中から「マクロ名」を探して「文字列」に置き換えます。

```
#define NO 5
```

と書くと、ソースファイル中のNOを5に置き換えます。単純な置き換えですが、この機能は強力です。

この置き換えを関数形式で行った場合、**関数形式マクロ**と呼ぶことがあります。

```
#define KAKEZAN(x,y) ((x)*(y))
```

とすると、KAKEZAN(2,3)が「2 * 3」に置き換わります。注意すべき点は、(x)、(y)というようにかっこをつけないと困ったことが起こりうる点です。

```
#define KAKEZAN(x,y) (x * y)
```

とすると、

```
int a = 2, b = 3, c = 4, d = 5, e;
e = KAKEZAN(a + b, c + d);
```

はどうなるでしょうか。a+bは5、c+dは9なので

```
e = (5 * 9);
e = 45;
```

となり、eに45が代入されることを期待されるのではないでしょうか。しかし、実際に置き換えてみると

```
e = (a + b * c + d);
```

となります。つまり、

```
e = (2 + 3 * 4 + 5);
e = 19;
```

となります。たいへんわかりにくいバグが発生しますので注意する必要があります。

```
#define 文字列 ……(1)
```

といった使い方もあります。そして、これと組み合わせて

```
#ifdef 文字列
……
#endif
```

の組み合わせで使います。(1)のように文字列が定義されていると、#ifdefと#endifの間の部分がコンパイルされます。(1)の定義がなければ#ifdef～#endifの部分は無視されます。また、これとは逆に

```
#ifndef 文字列
#endif
```

の組み合わせもあります。これは、文字列が定義されていないときに #ifndef 〜 #endif の部分がコンパイルされます。

#undef は #define 定義を無効にします。

```
#pragma 独自命令
```

というのもあります。コンパイラにより独自命令は異なります。ま、このくらい知っていればあまり不自由はしません。

11.8 時間関連の関数

プログラミングをしていると、現在の日付とか時間が知りたいことがよくあります。これは、どうしたらよいのでしょうか。

これには、time 関数を使います。

time 関数

システム時刻を取得します。

```
time_t time(
    time_t *timer
);
```

timer には、時刻の格納場所へのポインタを指定します。不要であれば NULL を指定してもかまいません。戻り値は、1970 年 1 月 1 日からの経過時間（秒）で、時差は考慮されていません。time_t 型は time.h の中で定義されています。

時差を考慮した時刻を取得するには localtime 関数を使います。

localtime 関数

時刻値を現地のタイムゾーンに合わせて変換します。

```
struct tm *localtime(
    const time_t *timer
);
```

timerには、格納されている時刻へのポインタを指定します。戻り値はtm型構造体へのポインタです。tm構造体はtime.hの中で次のように定義されています。

tm 構造体

時刻情報を格納します。

```
struct tm {
    int tm_sec;   /* 秒(0～59) */
    int tm_min;   /* 分(0～59) */
    int tm_hour;  /* 時(0～23) */
    int tm_mday;  /* 日(1～31) */
    int tm_mon;   /* 月(0～11) */
    int tm_year;  /* 年(実際の西暦から1900を引いた数) */
    int tm_wday;  /* 曜日(0～6、日曜日が0) */
    int tm_yday;  /* 年内の通算日(0～365、1月1日が0) */
    int tm_isdst; /* 夏時間の有効・無効を示すフラグ */
};
```

月は0～11で表されるので、使う際には1を足してやる必要があります。年については、1900を足すと西暦が求まります。

また、次のような関数も便利です。

asctime 関数

tm構造体を文字列に変換します。

```
char *asctime(
    const struct tm *timeptr
);
```

これは、tm構造体を文字列に変換します。
では、サンプルを作ってみましょう。

```c
/* time01.c */

#include <stdio.h>
#include <time.h>

int main()
{
    time_t orgtime;
    struct tm *lpmytm;
    char *today;

    orgtime = time(NULL);
    lpmytm = localtime(&orgtime);

    printf("現在時刻は%02d時%02d分%02d秒です¥n",
        lpmytm->tm_hour, lpmytm->tm_min, lpmytm->tm_sec);
    printf("今日は%d年%02d月%02d日です¥n",
        lpmytm->tm_year + 1900, lpmytm->tm_mon + 1,
        lpmytm->tm_mday);
    today = asctime(lpmytm);
    printf(today);

    return 0;
}
```

実行結果は次のようになります。

```
現在時刻は03時06分54秒です
今日は2008年01月16日です
Wed Jan 16 03:06:54 2008
```

11.9 Windows プログラミングに挑戦

さて、本書によりC言語の基礎的な知識が身についたならば、ぜひともWindowsのGUIアプリケーションをC言語でプログラミングすることに挑戦してください。

次にもっとも基本的なWindowsプログラムを紹介します。

```
// template.cpp

#include <windows.h>

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
ATOM InitApp(HINSTANCE);
BOOL InitInstance(HINSTANCE, int);

char szClassName[] = "template";    // ウィンドウクラス


int WINAPI WinMain(HINSTANCE hCurInst, HINSTANCE hPrevInst,
                   LPSTR lpsCmdLine, int nCmdShow)
{
    MSG msg;
    BOOL bRet;

    if (!InitApp(hCurInst))
        return FALSE;
    if (!InitInstance(hCurInst, nCmdShow))
        return FALSE;
    while ((bRet = GetMessage(&msg, NULL, 0, 0)) != 0) {
        if (bRet == -1) {
            break;
        } else {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
```

```
    return (int)msg.wParam;
}

// ウィンドウクラスの登録

ATOM InitApp(HINSTANCE hInst)
{
    WNDCLASSEX wc;
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WndProc;    // プロシージャ名
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInst;        // インスタンス
    wc.hIcon = (HICON)LoadImage(NULL,
        MAKEINTRESOURCE(IDI_APPLICATION),
        IMAGE_ICON,
        0,
        0,
        LR_DEFAULTSIZE | LR_SHARED);
    wc.hCursor = (HCURSOR)LoadImage(NULL,
        MAKEINTRESOURCE(IDC_ARROW),
        IMAGE_CURSOR,
        0,
        0,
        LR_DEFAULTSIZE | LR_SHARED);
    wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = NULL;      // メニュー名
    wc.lpszClassName = (LPCSTR)szClassName;
    wc.hIconSm = (HICON)LoadImage(NULL,
        MAKEINTRESOURCE(IDI_APPLICATION),
        IMAGE_ICON,
        0,
        0,
        LR_DEFAULTSIZE | LR_SHARED);

    return (RegisterClassEx(&wc));
}

// ウィンドウの生成
```

```
BOOL InitInstance(HINSTANCE hInst, int nCmdShow)
{
    HWND hWnd;

    hWnd = CreateWindow(szClassName,
            // タイトルバーにこの名前が表示されます
            "猫でもわかるWindowsプログラミング",
            WS_OVERLAPPEDWINDOW, // ウィンドウの種類
            CW_USEDEFAULT,       // X座標
            CW_USEDEFAULT,       // Y座標
            CW_USEDEFAULT,       // 幅
            CW_USEDEFAULT,       // 高さ
            NULL, // 親ウィンドウのハンドル、親を作るときはNULL
            NULL, // メニューハンドル、
                  //      クラスメニューを使うときはNULL
            hInst, // インスタンスハンドル
            NULL);
    if (!hWnd)
        return FALSE;
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    return TRUE;
}

// ウィンドウプロシージャ
LRESULT CALLBACK WndProc(
    HWND hWnd, UINT msg, WPARAM wp, LPARAM lp)
{
    int id;

    switch (msg) {
        case WM_CLOSE:
            id = MessageBox(hWnd,
                "終了してもよろしいですか",
                "確認",
                MB_YESNO | MB_ICONQUESTION);
            if (id == IDYES)
                DestroyWindow(hWnd);
            break;
        case WM_DESTROY:
```

```
            PostQuitMessage(0);
            break;
        default:
            return (DefWindowProc(hWnd, msg, wp, lp));
    }
    return 0;
}
```

なんだか難しそうなプログラムですね。

これを、今までのようにコンソールアプリケーションの設定でコンパイルしてもエラーになります。Visual C++ の場合はプロジェクトの設定を Windows アプリケーションにしてください。

典型的な Windows アプリケーションのウィンドウができたはずです（図 11-2）。ウィンドウの縁をドラッグすると、ウィンドウサイズを変更できます。タイトルバーをドラッグするとウィンドウを移動できます。最小化ボタン、最大化ボタン、閉じる（バツ）ボタンもついています。

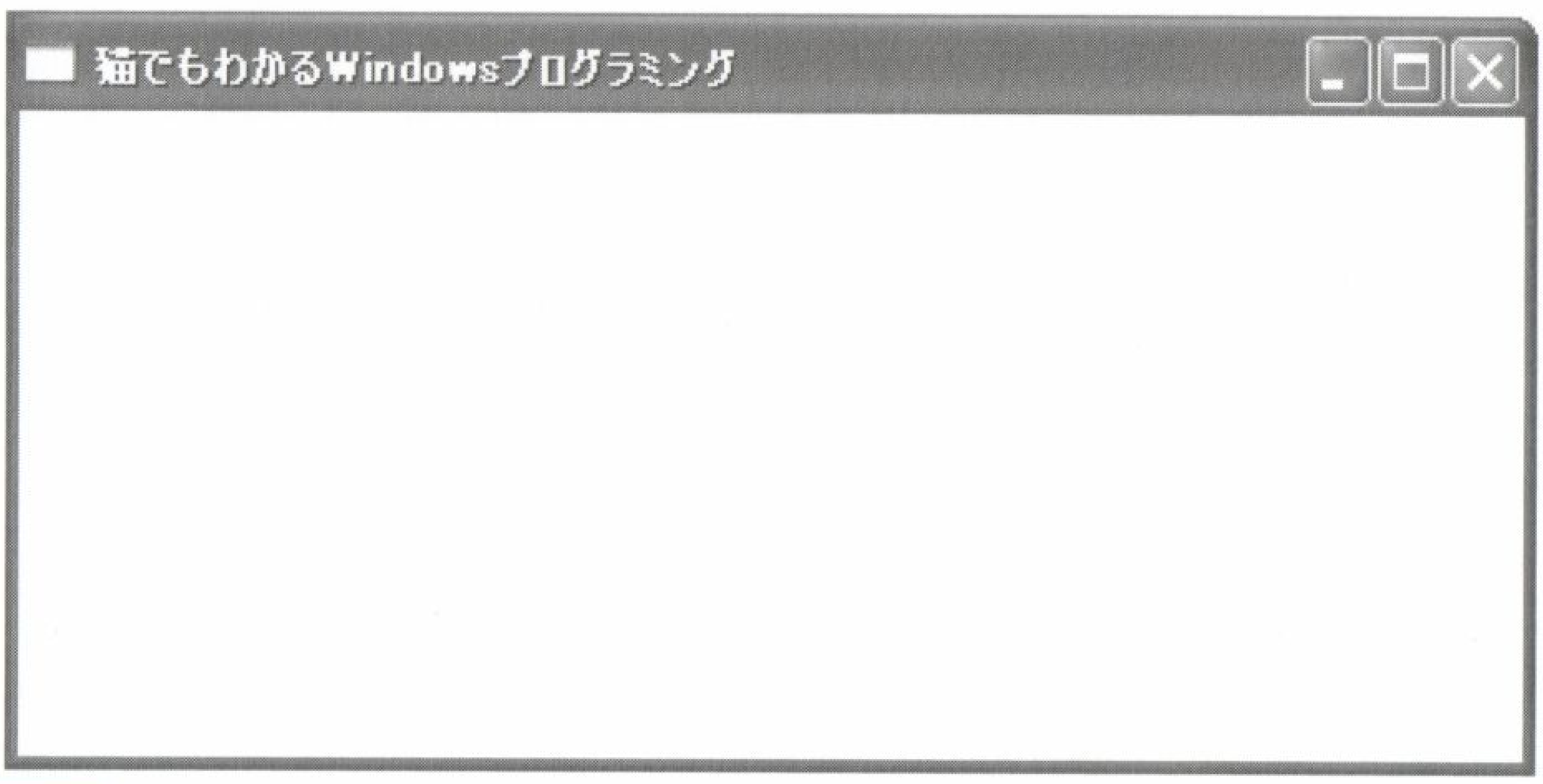


図 11-2　初めての Windows プログラム

一見難しそうに見えますが、この基本プログラムはどんなプログラムでもほとんど変わりません。変更したり、つけ足したりするのは、主に WndProc 関数のみです。

本書をここまで読み進んだ方は、すでにWindowsプログラミングを始める基礎的なプログラミング力がついているはずです。

練習問題

1. ユーザーの入力した文字(アスキー文字)を記憶するプログラムを作ってください。何も入力せずにEnterキーを押したら、入力した文字を順に表示するようにしてください。

※解答は付録CD-ROMに収録

第11章のまとめ

1. 関数はポインタを返すことができる。
2. 変数と同様、関数へのポインタも使うことができる。
3. 可変個の引数を持つ関数を作ることができる。
4. malloc関数・calloc関数で、メモリを動的に確保することができる。
5. realloc関数でメモリ領域の大きさを変更することができる。
6. ソースファイルは分割することができる。
7. #includeや#defineなどのプリプロセッサは、一番最初に処理される。
8. 時間関連の関数を使うには、<time.h>をインクルードする。

よくがんばりました…



付録A 開発環境の使い方

プログラムを作成するためには、開発環境（コンパイラ、リンカ、デバッガなど）を利用します。本書では、無償で利用できるMicrosoft Visual C++ 2008 Express Editionにおける作業手順を解説します（Visual C++のその他のバージョン、エディションもほぼ同様です）。

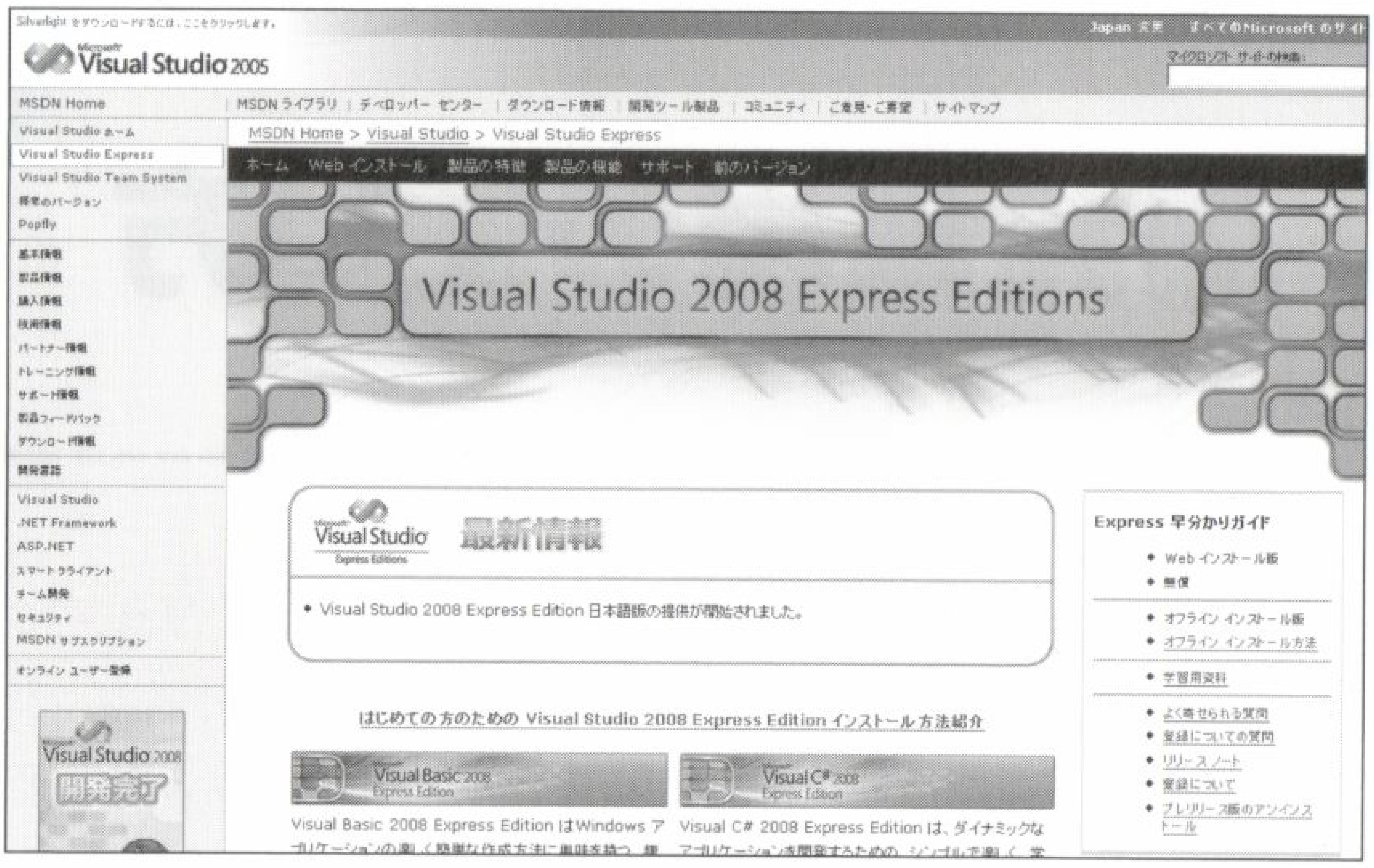
（注：この付録で紹介する手順はあくまでも一例です。ご使用の環境によって、手順や動作が多少異なる場合があります。詳細な製品情報や使い方、製品サポートにつきましては、開発元のマイクロソフト株式会社にお問い合わせください。）

A.1 Visual C++ 2008 Express Editionのインストール

Visual C++ 2008は、マイクロソフトが開発したC/C++の開発環境です。本書で扱ったC言語プログラムはもちろん、Windowsネイティブなアプリケーションも開発できます。いくつかのEditionがありますが、Express Edition

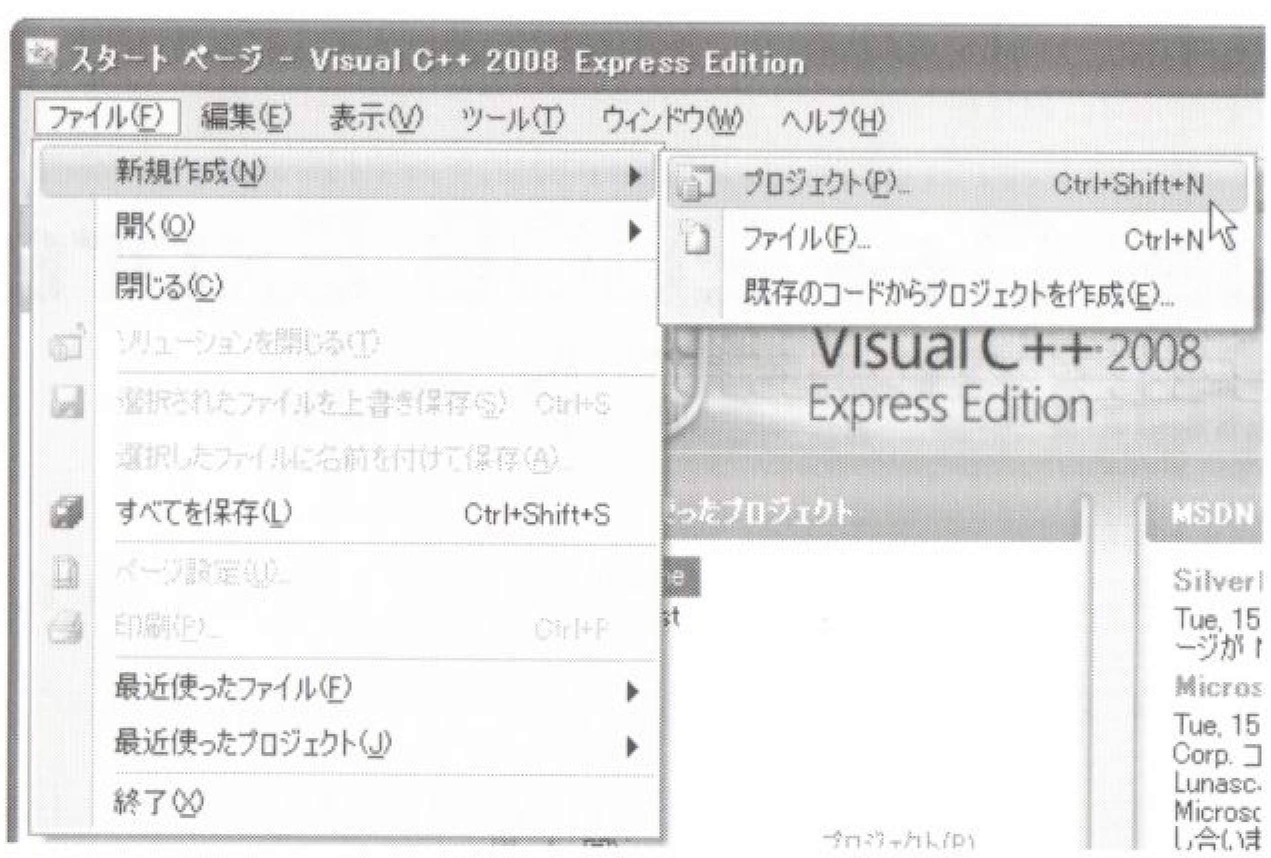
は評価、学習、ホビーに適しており、無償で利用できます。マイクロソフトのWebサイトからダウンロードして、インストールしてください。

- **Visual Studio 2008 Express Edition**：
  http://www.microsoft.com/japan/msdn/vstudio/express/



A.2 Visual C++の使い方

まずは、「ファイル」→「新規作成」から「プロジェクト」を選択します。

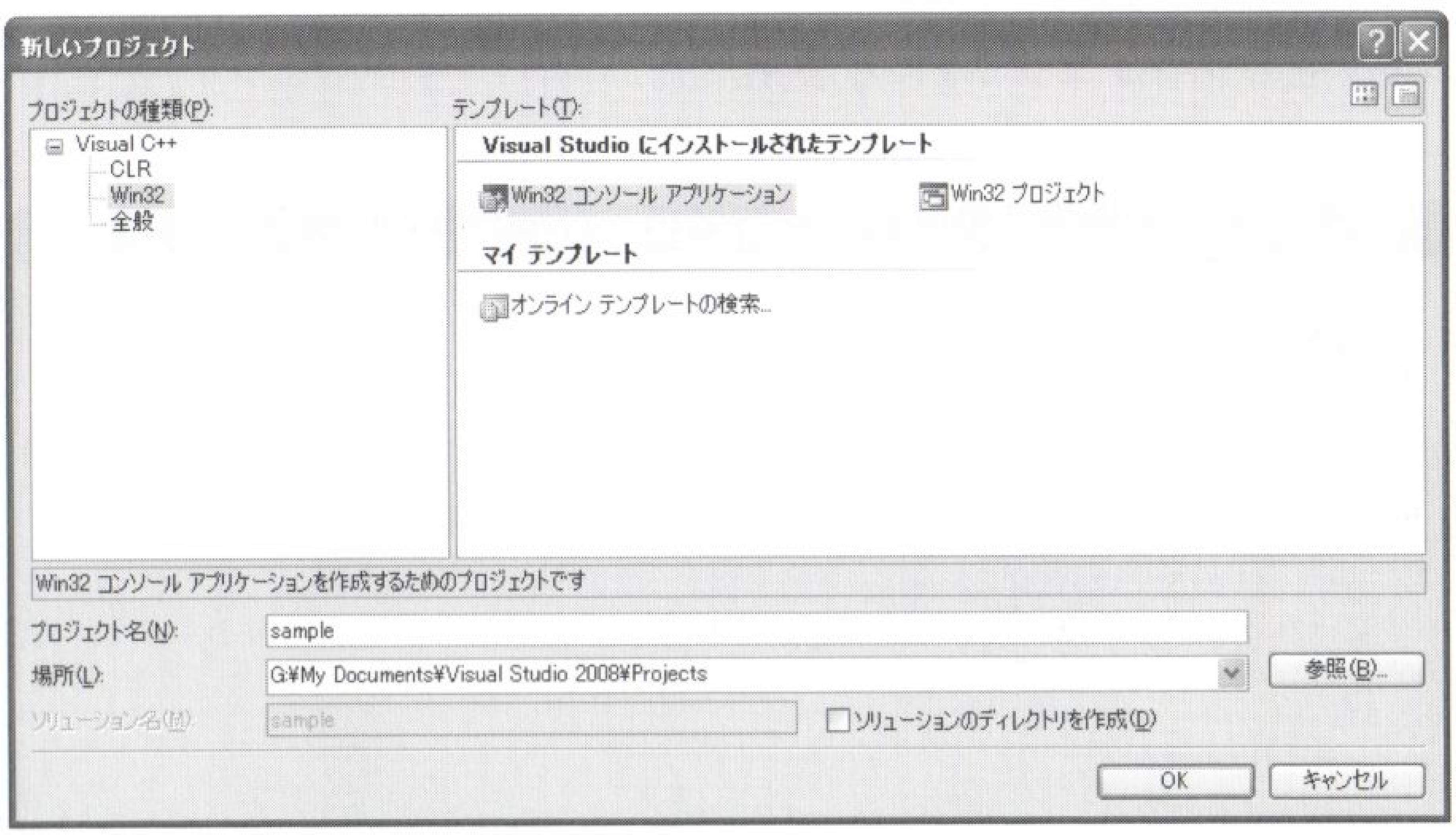


「新しいプロジェクト」ダイアログが出現するので、プロジェクトの種類を「Visual C++」の中の「Win32」に設定します。また、「Visual Studio にインストールされたテンプレート」を「Win32 コンソールアプリケーション」に設定します。

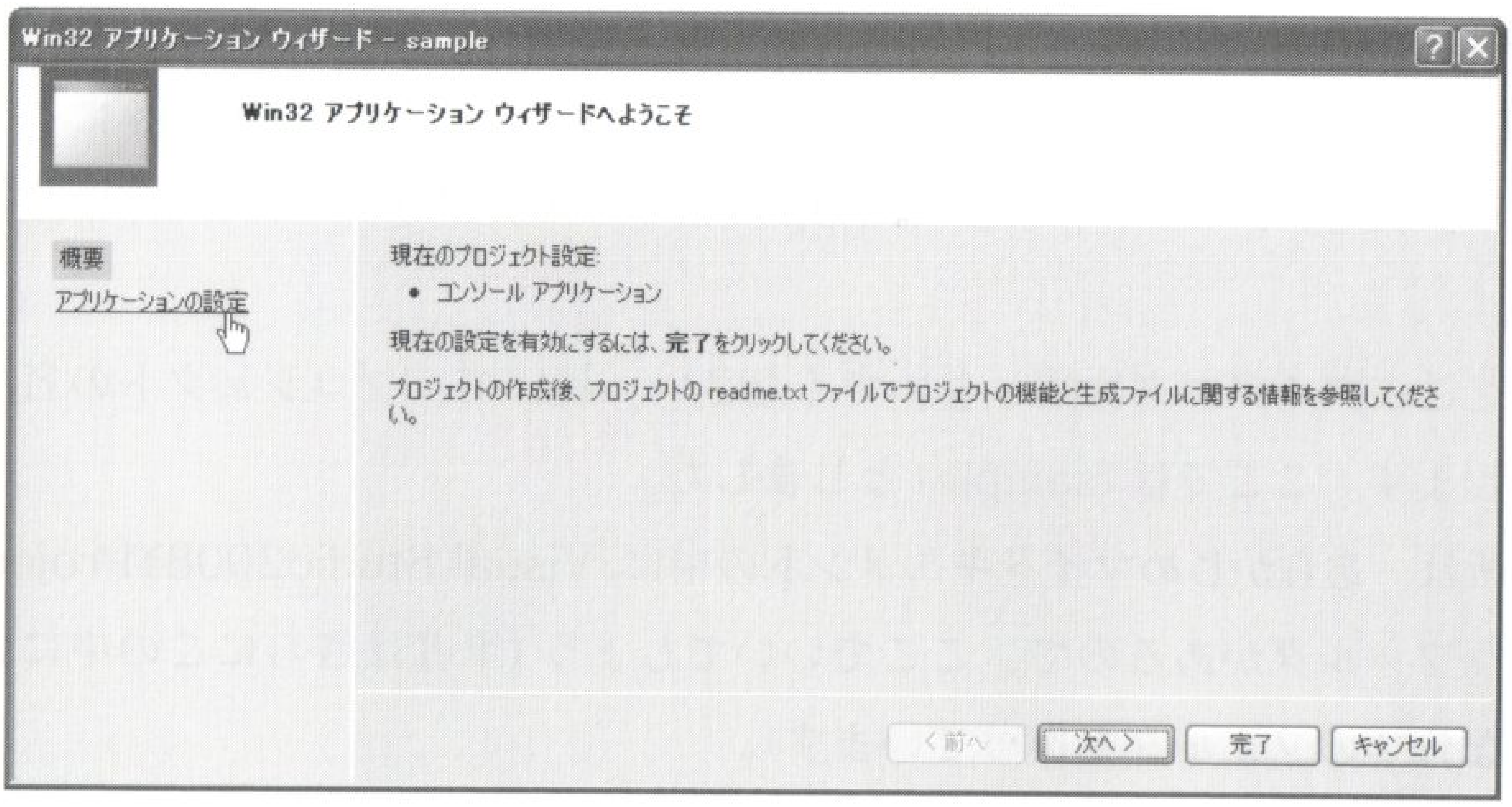
まだこのダイアログは閉じないでください。続いて、プロジェクトの名前を指定します。ここでは「sample」としました。

場所は、あらかじめマイドキュメントの中に「Visual Studio2008¥Projects」というフォルダがあるので、ここでいいでしょう（筆者はさらにこの中にいろいろな分類のフォルダを作っています）。

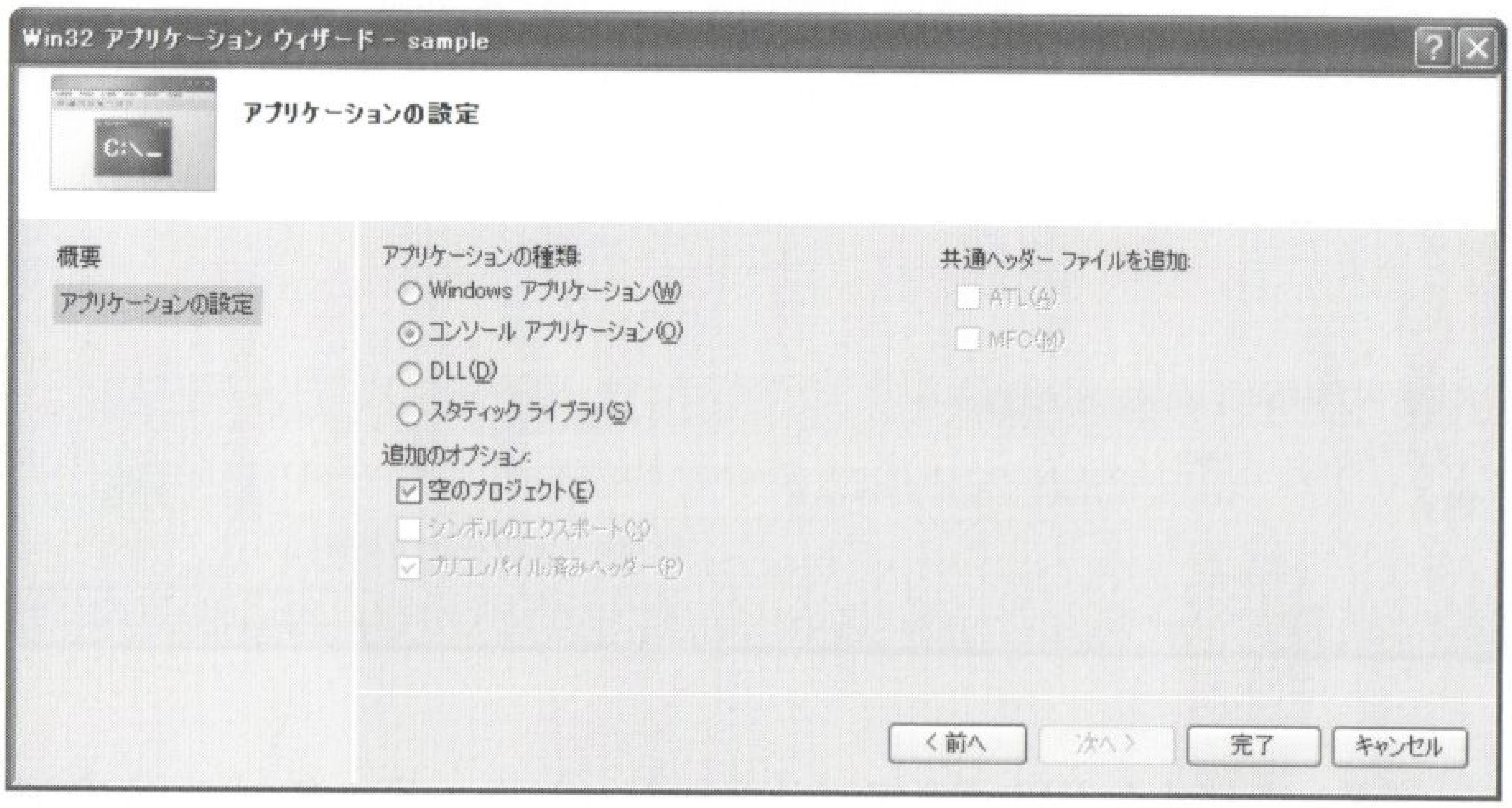
最初のうちは、1つのソリューションに1つのプロジェクトなので「ソリューションのディレクトリを作成」のチェックは外しておきます。



「OK」ボタンを押すと「Win32 アプリケーションウィザード」というのが出てきます。まずはこのウィザードの左側のリストから、「アプリケーションの設定」をクリックします。

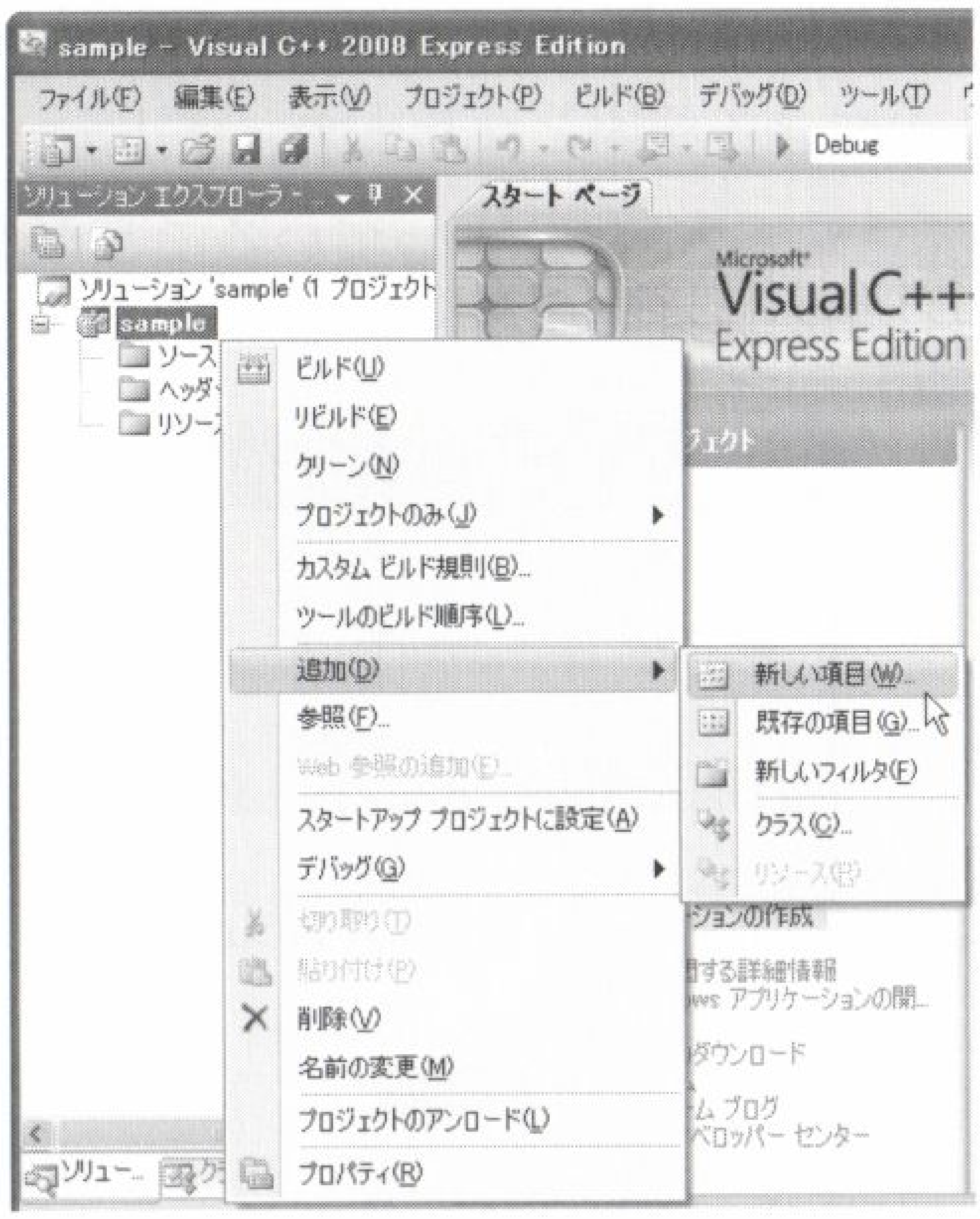


クリックすると、「アプリケーションの設定」画面に切り替わります。ここで、アプリケーションの種類を「コンソールアプリケーション」に、追加のオプションを「空のプロジェクト」にします。



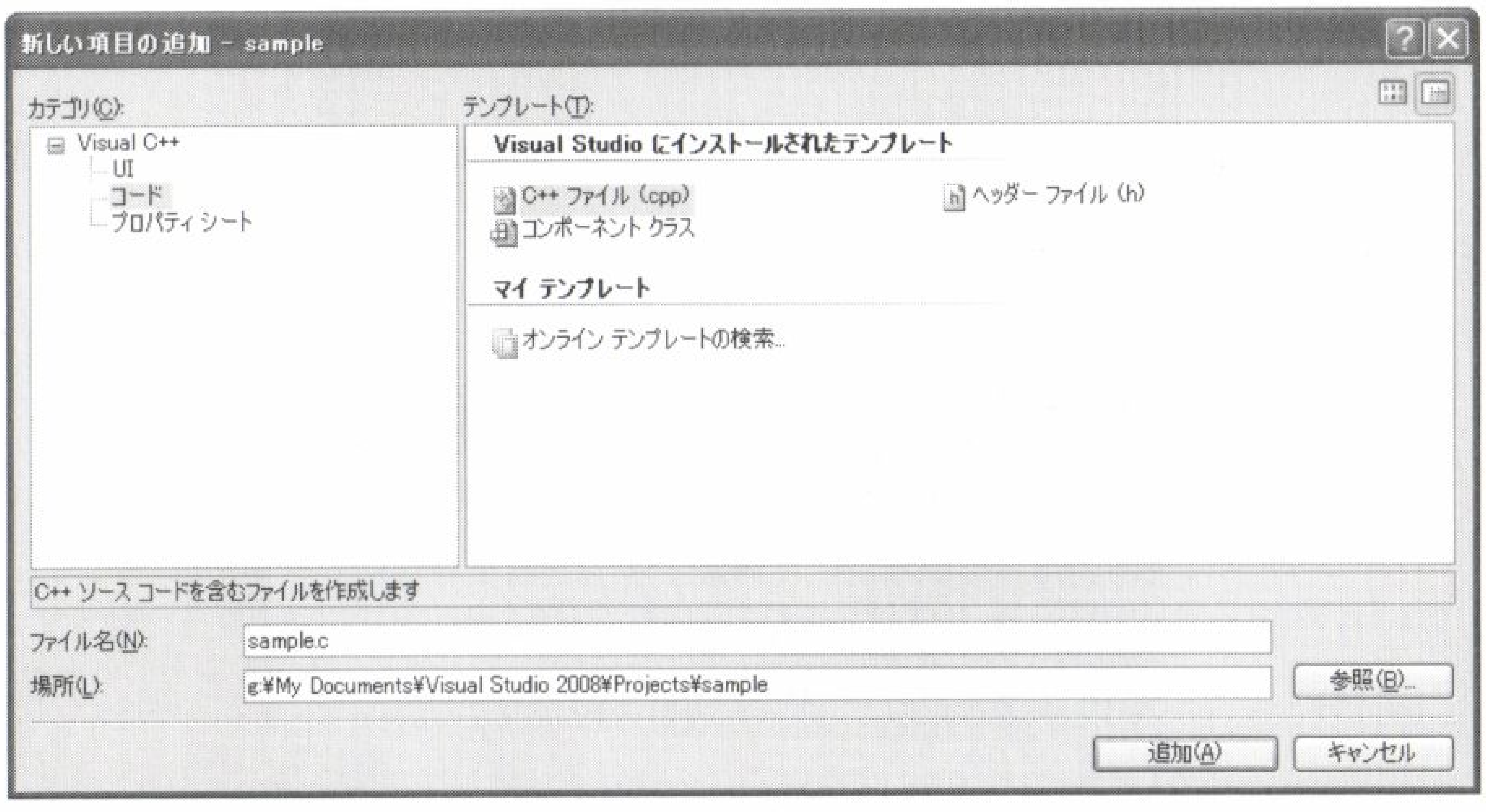
この状態で「完了」ボタンを押せばプロジェクトの設定は終了し、Visual C++ 2008のウィンドウに戻ります。しかし作成したのは空のプロジェクトなので、ソースファイルをプロジェクトに追加する必要があります。

Visual C++ 2008の左のほうにあるソリューションエクスプローラで、今作った「sample」を右クリックします。そして、表示されるコンテキストメニューから、「追加」→「新しい項目」を選択します。

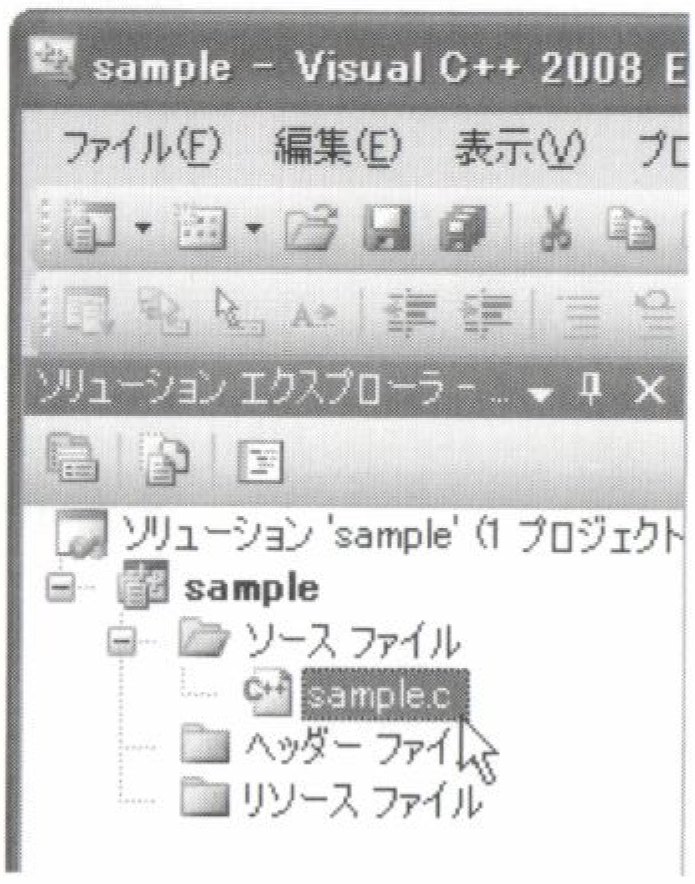


すると、「新しい項目の追加」ダイアログが出現します。ここで、ソースコードを書き込むためのファイルを指定します。カテゴリから「コード」を選択し、テンプレートから「C++ ファイル(cpp)」を選択してください。

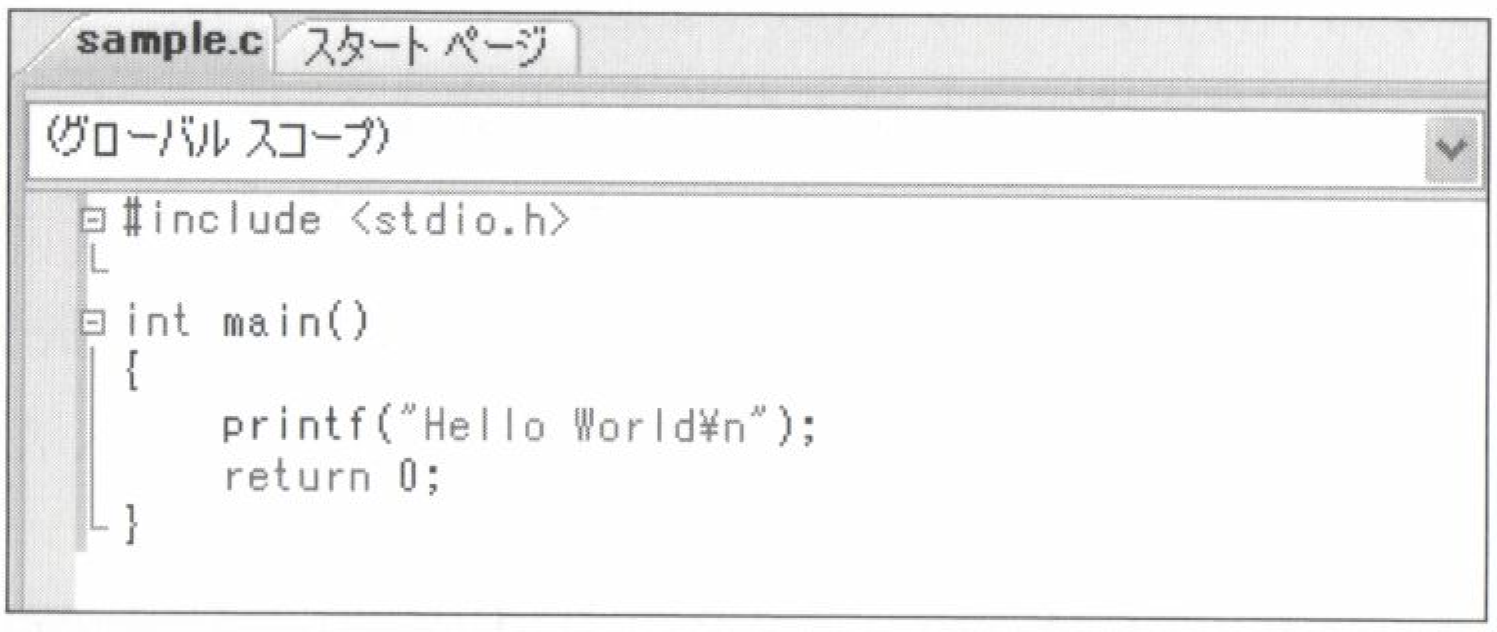
続いて、ファイル名の欄にファイル名を記入します。ソースファイル名をプロジェクト名と同じにしておくと便利です。ここではプロジェクト名が「sample」なので、ソースファイル名は「sample.c」とします。拡張子の「.c」まで入力しないと、自動的に「.cpp」となってしまうので注意してください。



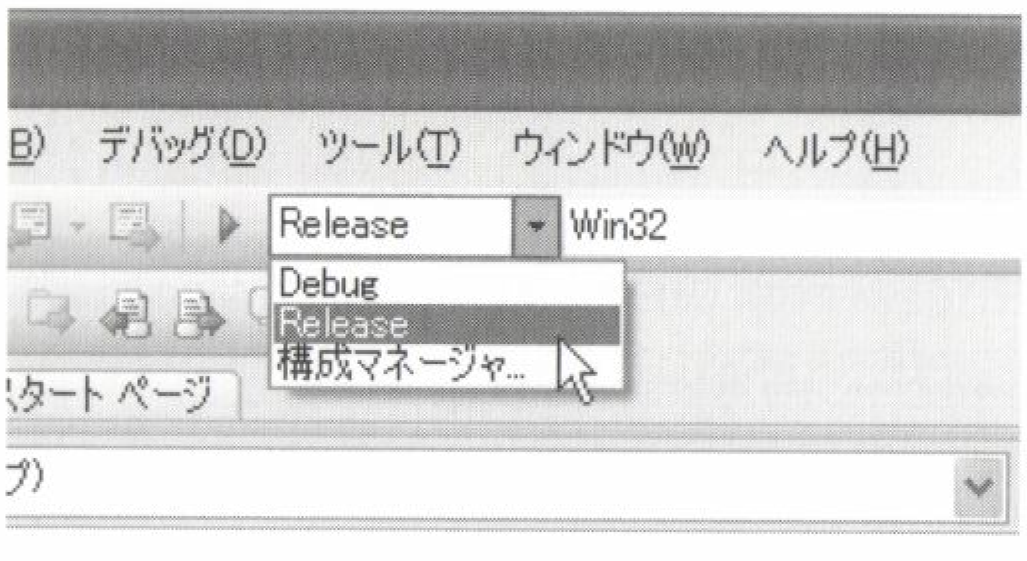
最後に「追加」ボタンを押します。Visual C++ 2008のウィンドウに戻り、すぐにプログラムを入力できる状態となります。ソリューションエクスプローラで、「sample.c」がプロジェクトに加わっていることを確認してください。



さあ、プログラムを書きましょう。



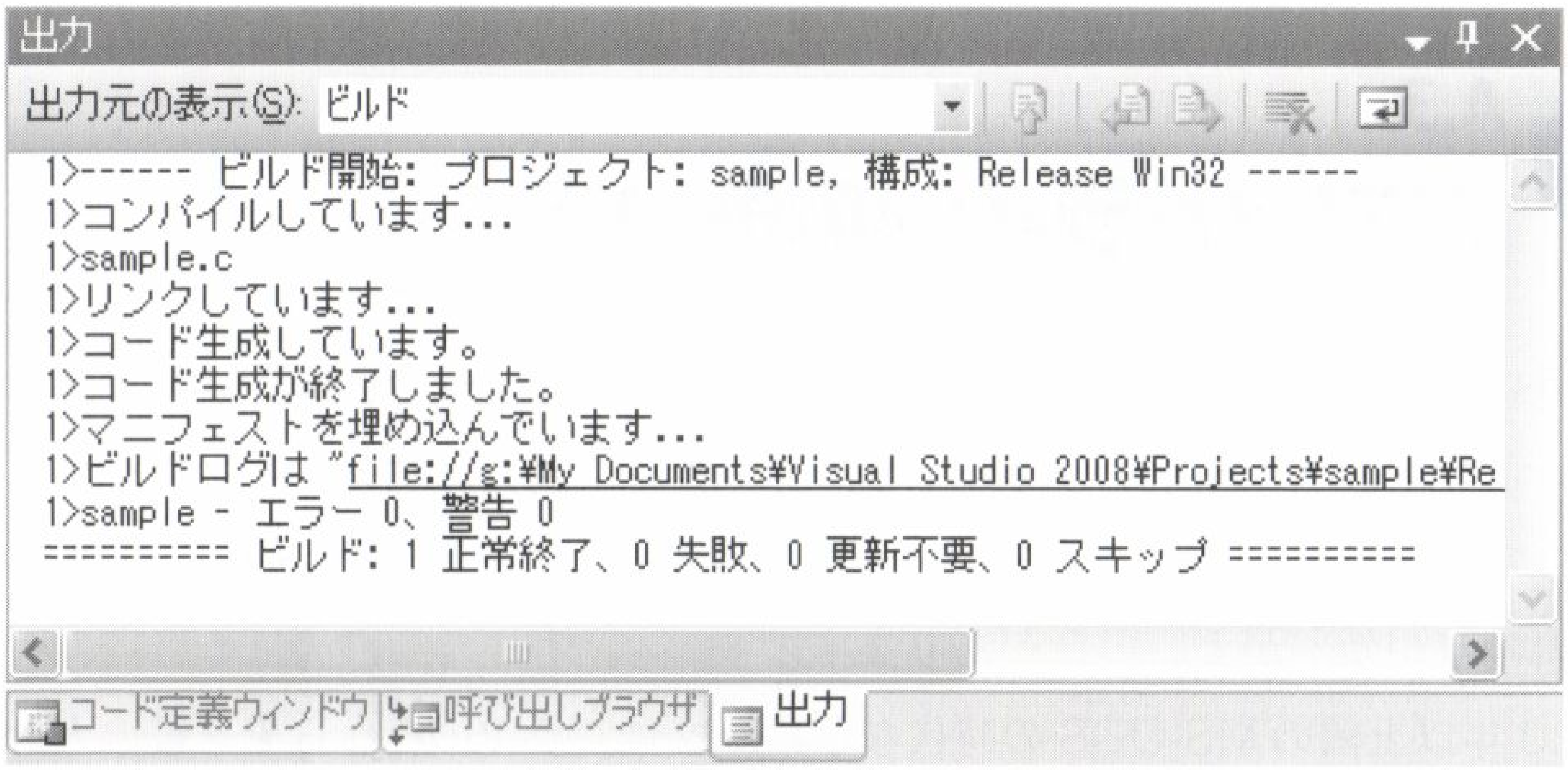
デバッガを使わないのであれば、ツールバーの「ソリューションの構成」を「Release」にしておきます。



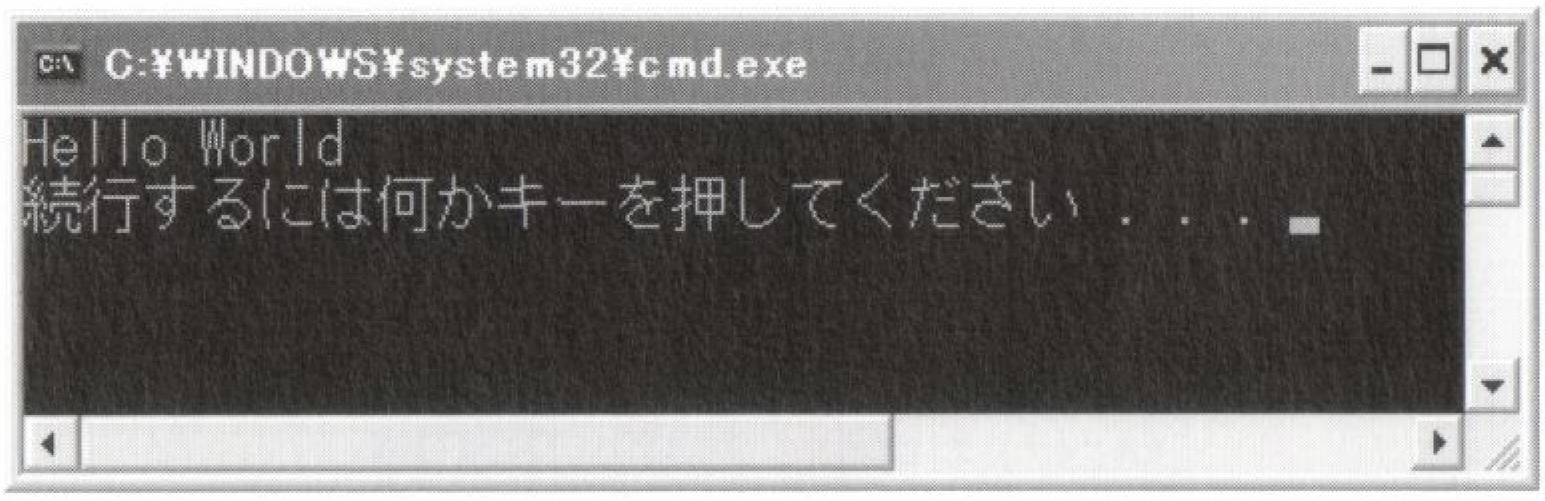
それではビルドしてみましょう。メニューから、「ビルド」→「sample のビルド」を選択します。



出力ウィンドウに「エラー 0、警告 0」と表示されたら大成功です。



いよいよテストランです。メニューから「デバッグ」→「デバッグなしで開始」を選択します。次の画面のようになったら、プログラムは正しく動いています。



セキュリティ強化バージョンの関数に関する補足

マイクロソフト社はVisual C++ 2005から、標準の関数名に「_s」という接尾辞をつけたセキュリティ強化バージョンの関数を導入しました。たとえばscanf関数の新バージョンは「scanf_s」関数となります。Visual C++ 2005/2008で標準の関数をそのまま使うと、警告が出ることがあります。警告なしでコンパイルするには、ソースプログラムの冒頭に、次のいずれかの行を追加してください。

```
#define _CRT_SECURE_NO_DEPRECATE
#pragma warning(disable : 4996)
```

しかし、これら新バージョンの関数は一般的に普及しているわけではなく、Visual C++ 2005以前の開発環境や、他社製コンパイラではコンパイルできません。

A.3 コマンドラインの使い方

Visual C++ 2008では、前節で使用したGUIツールの他に、コマンドラインによるコンパイル、リンクもできます。本書で扱ったサンプルプログラムはすべてコンソールに表示するコンソールアプリケーションですので、こちらの手順の方が効率がよい場合があります。

20年以上前のMS-DOSの時代などは、プログラマのみならず一般の人もコマンドライン上で作業していました。現在MS-DOS自体が動作しているPCはほとんど見かけませんが、Windows XP/Vistaでもコマンドラインで操作することができます。

コマンドラインは、次のいずれかの方法で起動できます。

- 「スタート」→「すべてのプログラム」→「アクセサリ」→コマンドプロンプト
- 「スタート」→「ファイル名を指定して実行」で「cmd」を起動

これでコマンドプロンプトが起動します。この画面にコマンドを打ち込んで操作します。

① コマンドラインよる操作

現在表示されているのがカレントディレクトリ名です。ディレクトリを移動する場合は、cd コマンドを使います。たとえば本書のサンプルプログラムが「C:¥sample」というディレクトリにある場合は、「**cd c:¥sample**」と入力することで移動できます。その他主なコマンドは表A-1に挙げましたが、詳細な使い方はWindowsのヘルプや解説書を参考にしてください。

表A-1　主なコマンド

コマンド名	動作
attrib	ファイル属性を変更する
cd	カレントディレクトリを変更する
cls	画面をクリアする
copy	ファイルをコピーする
del	ファイルを削除する
dir	ファイルの一覧を表示する
fc	ファイルを比較する
find	文字列を検索する
mkdir (md)	ディレクトリを作成する
rmdir (rd)	ディレクトリを削除する
rename	ファイル名を変更する
move	ファイルを移動する
set	環境変数を表示・設定する
type	ファイルの内容を表示する
xcopy	サブディレクトリを含めてファイルをコピーする

② コマンドラインによるコンパイル

C言語のソースファイルをコマンドラインでコンパイルするためには、コマンドラインにおける環境を設定する必要があります。Visual C++ 2008にはそのためのバッチファイルが用意されています。Visual C++ 2008をインストー

ルしたフォルダの中にある「VC¥bin¥vcvars32.bat」というバッチファイルを、コンパイルしたいソースファイルと同じフォルダにコピーしておいてください。

それではコンパイル作業の方法を見ていきましょう。C言語のソースファイルをコンパイルするには、cl.exeを使います。ここではC:¥sampleディレクトリにあるtest.cというソースファイルをコンパイルしてみます。太字の部分が入力する部分です。その他の部分は、プロンプトや表示されるメッセージです。

```
C:¥>cd c:¥sample
C:¥>sample>vcvars32

C:¥sample>"C:¥Program Files¥Microsoft Visual Studio 9.0¥Common7¥Tools¥
➡ vsvars32.bat"
Setting environment for using Microsoft Visual Studio 2008 x86 tools.

C:¥>sample>cl test.c
test.c
Microsoft(R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for
➡ 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

/out:test.exe
test.obj
```

c:¥sampleディレクトリに移動

バッチファイルを実行

test.cをコンパイル

これで「test.exe」という実行可能ファイルが作成されました。コマンドラインで「test」と入力してEnterキーを押すと、このプログラムが実行されます。



付録B

標準ライブラリ関数一覧

本書で使用したC言語関数の一覧です。この他にもたくさんの関数が存在しますので、詳細はMSDN等で確認してください。

■ファイル・ストリーム関連

関数	機能	ページ
fclose 関数	ファイルを閉じます。	249
fgetc 関数	ファイルから1文字ずつ読み込みます。	259
fopen 関数	ファイルを開きます。	248
fprintf 関数	ストリームに整形済みデータを書き込みます。	249
fread 関数	ストリームからデータを読み込みます。	276
fscanf 関数	ストリームから整形済みデータを読み込みます。	255
fseek 関数	ファイルポインタを指定した場所に移動します。	260
fwrite 関数	ストリームにデータを書き込みます。	275
getchar 関数	標準入力から文字を読み込みます。	106
gets 関数	標準入力から文字列を読み込みます。	197
printf 関数	標準出力に整形済みデータを出力します。	297
putc 関数	ファイルに文字を出力します。	259
scanf 関数	標準入力から整形済みデータを読み込みます。	92

■文字列関連

関数	機能	ページ
atof 関数	文字列を浮動小数点数に変換します。	232
atoi 関数	文字列を整数に変換します。	232
sprintf 関数	書式にのっとって変換した出力を文字列に格納します。	202
strcat 関数	文字列を連結します。	201
strcmp 関数	2つの文字列を比較します。	190
strcpy 関数	文字列をコピーします。	200
strlen 関数	文字列の長さを返します。	191
strstr 関数	文字列を検索します。	269

■メモリ関連

関数	機能	ページ
calloc 関数	メモリ配列を確保し、0で初期化します。	303
free 関数	メモリを解放します。	299
malloc 関数	メモリを取得します。	298
memset 関数	指定した文字でバッファを埋め尽くします。	236
realloc 関数	メモリの再確保を行います。	305

■日付・時間関連

関数	機能	ページ
asctime 関数	tm 構造体を文字列に変換します。	315
localtime 関数	時刻値を現地のタイムゾーンに合わせて変換します。	315
time 関数	システム時刻を取得します。	314
tm 構造体	時刻情報を格納します。	315

■その他

関数	機能	ページ
exit 関数	プログラムを正常に終了させます。	307
perror 関数	エラーメッセージを出力します。	301

付録C

ASCIIコード表

ASCIIとはAmerican Standard Code for Information Interchangeの略で、コンピュータで英数字･記号を扱うためのコード体系としてもっとも広く使われています。

この付録では、C言語プログラムで使いやすいように、10進、8進、16進でコードを示しています。

ASCIIコードには、実際に画面に表示される英数字･記号のほかに、改行やバックスペースなどの制御文字も含まれています。次のページの表のうち、グレーの部分が制御文字です。代表的な制御文字には次のものがあります。

表C-1　代表的なASCII制御文字

制御文字	コード	意味
NUL	0x00	ヌル(空文字)
BEL	0x07	ベル
BS	0x08	バックスペース
HT	0x09	水平タブ
NL	0x0a	改行
VT	0x0b	垂直タブ

制御文字	コード	意味
NP	0x0c	改ページ
CR	0x0d	復帰
ESC	0x1b	エスケープ
SP	0x20	スペース
DEL	0x7f	削除

表C-2 ASCIIコード表

文字	コード		
	10進	8進	16進
NUL	0	0	0x00
SOH	1	1	0x01
STX	2	2	0x02
ETX	3	3	0x03
EOT	4	4	0x04
ENQ	5	5	0x05
ACK	6	6	0x06
BEL	7	7	0x07
BS	8	10	0x08
HT	9	11	0x09
NL	10	12	0x0a
VT	11	13	0x0b
NP	12	14	0x0c
CR	13	15	0x0d
SO	14	16	0x0e
SI	15	17	0x0f
DLE	16	20	0x10
DC1	17	21	0x11
DC2	18	22	0x12
DC3	19	23	0x13
DC4	20	24	0x14
NAK	21	25	0x15
SYN	22	26	0x16
ETB	23	27	0x17
CAN	24	30	0x18
EM	25	31	0x19
SUB	26	32	0x1a
ESC	27	33	0x1b
FS	28	34	0x1c
GS	29	35	0x1d
RS	30	36	0x1e
US	31	37	0x1f

文字	コード		
	10進	8進	16進
SP	32	40	0x20
!	33	41	0x21
"	34	42	0x22
#	35	43	0x23
$	36	44	0x24
%	37	45	0x25
&	38	46	0x26
'	39	47	0x27
(	40	50	0x28
)	41	51	0x29
*	42	52	0x2a
+	43	53	0x2b
,	44	54	0x2c
-	45	55	0x2d
.	46	56	0x2e
/	47	57	0x2f
0	48	60	0x30
1	49	61	0x31
2	50	62	0x32
3	51	63	0x33
4	52	64	0x34
5	53	65	0x35
6	54	66	0x36
7	55	67	0x37
8	56	70	0x38
9	57	71	0x39
:	58	72	0x3a
;	59	73	0x3b
<	60	74	0x3c
=	61	75	0x3d
>	62	76	0x3e
?	63	77	0x3f

文字	コード		
	10進	8進	16進
@	64	100	0x40
A	65	101	0x41
B	66	102	0x42
C	67	103	0x43
D	68	104	0x44
E	69	105	0x45
F	70	106	0x46
G	71	107	0x47
H	72	110	0x48
I	73	111	0x49
J	74	112	0x4a
K	75	113	0x4b
L	76	114	0x4c
M	77	115	0x4d
N	78	116	0x4e
O	79	117	0x4f
P	80	120	0x50
Q	81	121	0x51
R	82	122	0x52
S	83	123	0x53
T	84	124	0x54
U	85	125	0x55
V	86	126	0x56
W	87	127	0x57
X	88	130	0x58
Y	89	131	0x59
Z	90	132	0x5a
[	91	133	0x5b
\	92	134	0x5c
]	93	135	0x5d
^	94	136	0x5e
_	95	137	0x5f

<table>
<tr><th rowspan="2">文字</th><th colspan="3">コード</th></tr>
<tr><th>10進</th><th>8進</th><th>16進</th></tr>
<tr><td>`</td><td>96</td><td>140</td><td>0x60</td></tr>
<tr><td>a</td><td>97</td><td>141</td><td>0x61</td></tr>
<tr><td>b</td><td>98</td><td>142</td><td>0x62</td></tr>
<tr><td>c</td><td>99</td><td>143</td><td>0x63</td></tr>
<tr><td>d</td><td>100</td><td>144</td><td>0x64</td></tr>
<tr><td>e</td><td>101</td><td>145</td><td>0x65</td></tr>
<tr><td>f</td><td>102</td><td>146</td><td>0x66</td></tr>
<tr><td>g</td><td>103</td><td>147</td><td>0x67</td></tr>
<tr><td>h</td><td>104</td><td>150</td><td>0x68</td></tr>
<tr><td>i</td><td>105</td><td>151</td><td>0x69</td></tr>
<tr><td>j</td><td>106</td><td>152</td><td>0x6a</td></tr>
<tr><td>k</td><td>107</td><td>153</td><td>0x6b</td></tr>
<tr><td>l</td><td>108</td><td>154</td><td>0x6c</td></tr>
<tr><td>m</td><td>109</td><td>155</td><td>0x6d</td></tr>
<tr><td>n</td><td>110</td><td>156</td><td>0x6e</td></tr>
<tr><td>o</td><td>111</td><td>157</td><td>0x6f</td></tr>
<tr><td>p</td><td>112</td><td>160</td><td>0x70</td></tr>
<tr><td>q</td><td>113</td><td>161</td><td>0x71</td></tr>
<tr><td>r</td><td>114</td><td>162</td><td>0x72</td></tr>
<tr><td>s</td><td>115</td><td>163</td><td>0x73</td></tr>
<tr><td>t</td><td>116</td><td>164</td><td>0x74</td></tr>
<tr><td>u</td><td>117</td><td>165</td><td>0x75</td></tr>
<tr><td>v</td><td>118</td><td>166</td><td>0x76</td></tr>
<tr><td>w</td><td>119</td><td>167</td><td>0x77</td></tr>
<tr><td>x</td><td>120</td><td>170</td><td>0x78</td></tr>
<tr><td>y</td><td>121</td><td>171</td><td>0x79</td></tr>
<tr><td>z</td><td>122</td><td>172</td><td>0x7a</td></tr>
<tr><td>{</td><td>123</td><td>173</td><td>0x7b</td></tr>
<tr><td>|</td><td>124</td><td>174</td><td>0x7c</td></tr>
<tr><td>}</td><td>125</td><td>175</td><td>0x7d</td></tr>
<tr><td>~</td><td>126</td><td>176</td><td>0x7e</td></tr>
<tr><td>DEL</td><td>127</td><td>177</td><td>0x7f</td></tr>
</table>

索引

記号・数字

A

く, け

こ

さ, し

す

せ

そ

た

て，と

な行

は

ひ

ふ

へ

ほ

み，む

め

も

よ

ら行

わ

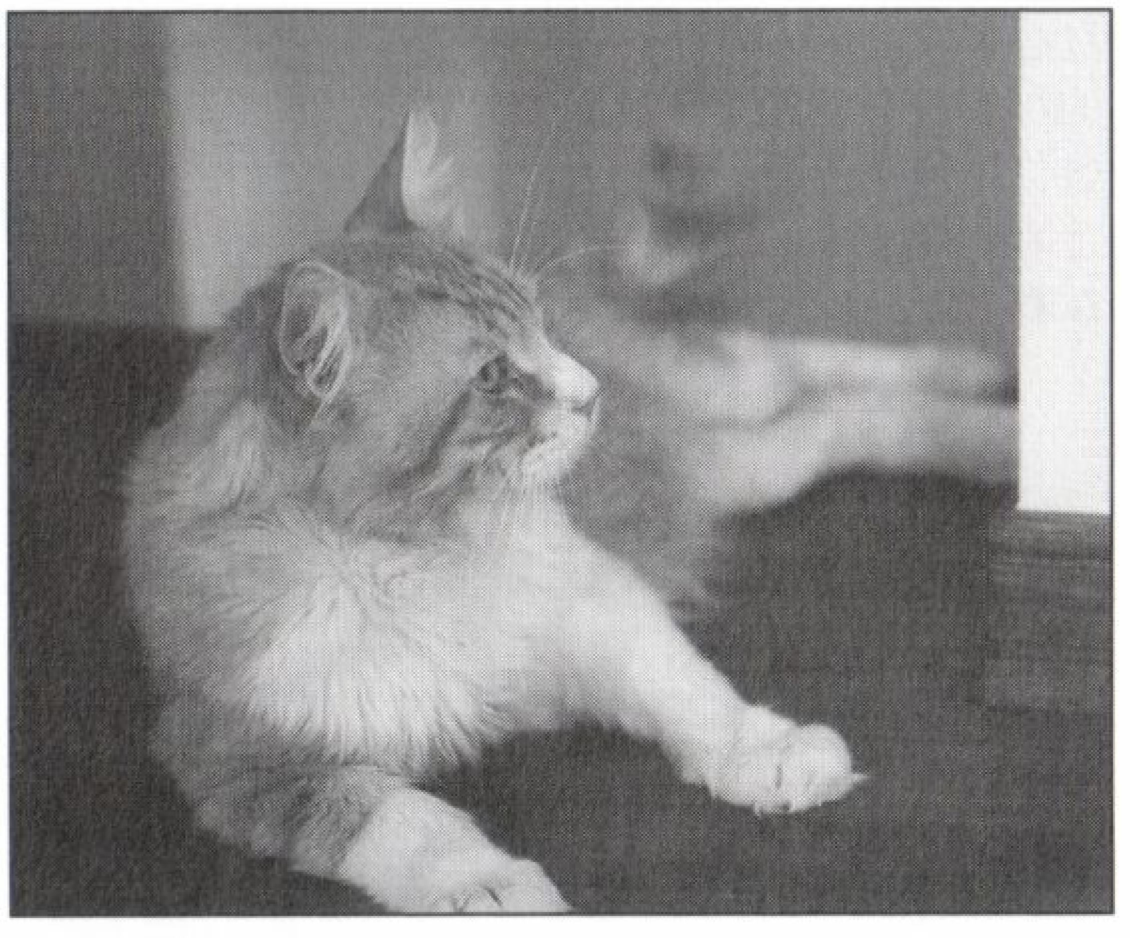
●著者紹介

条井 康孝（くめい やすたか）

旭川医科大学卒業。糖尿病専門医。
現在、深川第一病院副院長。
シェアウェア作家協会会員。「条井ネット」代表。

SQL Server + C言語によるクライアントでデータベースシステムを開発し、実際の業務に活用して数年経過した。やはり、C言語で書かれたソフトは速い。Visual Basicによるクライアントに比べ、そのスピードは自転車とスペースシャトルほどの違いがあると実感している。

E-Mail: yasutaka@kumei.ne.jp
URL: http://www.kumei.ne.jp/



『猫でもわかるC言語プログラミング　第2版』サポートページ

http://isbn.sbcr.jp/45650/

お電話でのお問い合わせにはいっさいお答えできませんので、なにとぞご了承ください。Webサイトにアクセスする手段をおもちでない方は、ご氏名、ご送付先を明記の上、下記宛先までお問い合わせください。

送付先住所　〒106-0032 東京都港区六本木2-4-5
FAX番号　03-5549-1144
宛先　ソフトバンク クリエイティブ株式会社
情報書籍編集部 読者サポート係

ただし、本書の記載内容とは直接関係のない一般的なご質問、本書の記載内容以上の詳細なご質問、お客様固有の環境に起因する問題についてのご質問、書籍内にすでに回答が記載されているご質問、具体的な内容を特定できないご質問など、そのご質問への対応が、他のお客様ならびに関係各位の権益を減損しかねないと判断される場合には、ご対応をお断りせざるをえないこともあります。またご質問の内容によっては、回答に数日ないしそれ以上の期間を要する場合もありますので、なにとぞご了承ください。

猫でもわかるC言語プログラミング　第2版

2004年4月17日　初版 第1刷 発行
2007年10月1日　初版 第12刷 発行
2008年3月6日　第2版 第1刷 発行
2012年9月3日　第2版 第10刷 発行

著　者　粂井 康孝（くめい やすたか）
発行者　新田 光敏
発行所　ソフトバンク クリエイティブ株式会社
〒106-0032 東京都港区六本木2-4-5
http://www.sbcr.jp/
制　作　編集マッハ
印　刷　文唱堂印刷株式会社

カバーデザイン　teeth

乱丁本、落丁本は小社営業部にてお取り替えいたします。
定価はカバーに記載されております。

Printed in Japan　ISBN978-4-7973-4565-0

ISBN978-4-7973-4565-0
C0055 ¥1900E

9784797345650

定価 本体1,900円 +税

1920055019002

SoftBank Creative

NEKO series

猫でもわかるC言語プログラミング 第2版

粂井康孝 著
Yasutaka Kumei

SoftBank Creative


NEKO series
猫でもわかる
C言語
プログラミング
第2版
SoftBank Creative
©2008 SOFTBANK
Creative Corp.
SBP4565
Made in Japan
収録内容
■C言語サンプルコード
■章末練習問題の解答
COMPACT disc